Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors

Anant Agarwal, David A. Kranz, and Venkat Natarajan

***Abstract*—This paper presents a theoretical framework for automatically partitioning parallel loops to minimize cache coherency traffic on shared-memory multiprocessors. While several previous papers have looked at hyperplane partitioning of iteration spaces to reduce communication traffic, the problem of deriving the *optimal* tiling parameters for minimal communication in loops with general affine index expressions has remained open. Our paper solves this open problem by presenting a method for deriving an optimal hyperparallelepiped tiling of iteration spaces for minimal communication in multiprocessors with caches. We show that the same theoretical framework can also be used to determine optimal tiling parameters for both data and loop partitioning in distributed memory multicomputers. Our framework uses matrices to represent iteration and data space mappings and the notion of uniformly intersecting references to capture temporal locality in array references. We introduce the notion of data footprints to estimate the communication traffic between processors and use linear algebraic methods and lattice theory to compute precisely the size of data footprints. We have implemented this framework in a compiler for Alewife, a distributed shared-memory multiprocessor.**

***Index Terms*—Automatic loop partitioning, shared-memory multiprocessors, compilers, tiling, minimizing communication.**

I. INTRODUCTION

CACHE-BASED multiprocessors are attractive because they seem to allow the programmer to ignore the issues of data partitioning and placement. Because caches dynamically copy data close to where it is needed, repeat references to the same piece of data do not require communication over the network, and hence reduce the need for careful data layout. However, the performance of cache-coherent systems is heavily predicated on the degree of temporal locality in the access patterns of the processor. Loop partitioning for cache-coherent multiprocessors is an effort to increase the percentage of references that hit in the cache.

The degree of reuse of data, or conversely, the volume of communication of data, depends both on the algorithm and on the partitioning of work among the processors. (In fact, partitioning of the computation is often considered to be a facet of an algorithm.) For example, it is well known that a matrix multiply computation distributed to the processors by square blocks has a much higher degree of reuse than the matrix multiply distributed by rows or columns.

Loop partitioning can be done by the programmer, by the run time system, or by the compiler. Relegating the partitioning task to the programmer defeats the central purpose of building cache-coherent shared-memory systems. While partitioning can be done at run time (for example, see [1], [2]), it is hard for the run time system to optimize for cache locality because much of the information required to compute communication patterns is either unavailable at run time or expensive to obtain. Thus compile-time partitioning of parallel loops is important.

This paper focuses on the following problem in the context of cache-coherent multiprocessors. Given a program consisting of parallel do loops (of the form shown in Fig. 1 in Section II.A), how do we derive the optimal tile shapes of the iteration-space partitions *to minimize the communication traffic* between processors. We also indicate how our framework can be used for loop and data partitioning for distributed memory machines, both with and without caches.

A. Contributions and Related Work

This paper develops a unified theoretical framework that can be used for loop partitioning in cache-coherent multiprocessors, or for loop and data partitioning in multicomputers with local memory.[1] The central contribution of this paper is a method for deriving an optimal hyperparallelepiped tiling of iteration spaces to minimize communication. The tiling specifies both the shape and size of iteration space tiles. Our framework allows the partitioning of doall loops accessing multiple arrays, where the index expressions in array accesses can be any affine function of the indices.

Our analysis uses the notion of uniformly intersecting references to categorize the references within a loop into classes that will yield cache locality. This notion helps specify precisely the set of references that have substantially overlapping data sets. Overlap produces temporal locality in cache accesses. A similar concept of uniformly generated references has been used in earlier work in the context of *reuse* and iteration space tiling [3], [4].

The notion of data footprints is introduced to capture the combined set of data accesses made by references within each uniformly intersecting class. (The term *footprint* was originally

Manuscript received Aug. 31, 1993; revised Apr. 26, 1994.
A short version of this paper appears in the *Proceedings of ICPP* 1993.
A. Agarwal and D.A. Kranz are with the MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139; e-mail: agarwal@mit.edu.
V. Natarajan is with the Motorola Cambridge Research Center, One Kendall Square, Bldg. 600, Cambridge, MA 02139.
IEEECS Log Number D95034.

1. This paper, however, focuses on loop partitioning, but indicates the modifications necessary for data partitioning. See [15] for results on combined loop and data partitioning.

coined by Stone and Thiebaut [5].) Then, an algorithm to compute precisely the total size of the data footprint for a given loop partition is presented. Precisely computing the size of the set of data elements accessed by a loop tile was itself an important and open problem. While general optimization methods can be applied to minimize the size of the data footprint and derive the corresponding loop partitions, we demonstrate several important special cases where the optimization problem is very simple. The size of data footprints can also be used to guide program transformations to achieve better cache performance in uniprocessors as well.

Although there have been several papers on hyperplane partitioning of iteration spaces, the problem of deriving the optimal hyperparallelepiped tile parameters for general affine index expressions has remained open. For example, Irigoin and Triolet [6] introduce the notion of loop partitioning with multiple hyperplanes which results in hyperparallelepiped tiles. The purpose of tiling in their case is to provide parallelism across tiles, and vector processing and data locality within a tile. They propose a set of basic constraints that should be met by any partitioning and derive the conditions under which the hyperplane partitioning satisfies these constraints.

Although their paper describes useful properties of hyperplane partitioning, it does not address the issue of automatically generating the tile parameters. Careful analysis of the mapping from the iteration space to the data space is very important in automating the partitioning process. Our paper describes an algorithm for automatically computing the partition based on the notion of cumulative footprints, derived from the mapping from iteration space to data space.

Abraham and Hudak [7] considered loop partitioning in multiprocessors with caches. However, they dealt only with index expressions of the form index variable plus a constant. They assumed that the array dimension was equal to the loop nesting and focused on rectangular and hexagonal tiles. Furthermore, the code body was restricted to an update of $A[i, j]$.

Our framework, however, does not place these restrictions on the code body. It is able to handle much more general index expressions, and produce parallelogram partitions if desired. We also show that when Abraham and Hudak's methods can be applied to a given loop nest, our theoretical framework reproduces their results.

Ramanujam and Sadayappan [8] deal with data partitioning in multicomputers with local memory and use a matrix formulation; their results do not apply to multiprocessors with caches. Their theory produces communication-free hyperplane partitions for loops with affine index expressions when such partitions exist. However, when communication-free partitions do not exist, they can deal only with index expression of the form variable plus a constant offset. They further require the array dimension to be equal to the loop nesting.

In contrast, our framework is able to discover optimal partitions in cases where communication free partitions are not possible, and we do not restrict the loop nesting to be equal to array dimension. In addition, we show that our framework correctly produces partitions identical to those of Ramanujam and Sadayappan when communication-free partitions do exist.

In a recent paper, Anderson and Lam [9] derive communication-free partitions for multicomputers when such partitions exist, and block loops into squares otherwise. Our notion of cumulative footprints allows us to derive optimal partitions even when communication-free partitions do not exist.

Gupta and Banerjee [10] address the problem of automatic data partitioning by analyzing the entire program. Although our paper deals with loop and data partitioning for a single loop only, the following differences in the machine model and the program model lead to problems that are not addressed by Gupta and Banerjee:

1) The data distributions considered by them do not include general hyperparallelepipeds. In order to deal with hyperparallelepipeds, one requires the analysis of communication presented in our paper.
2) Their communication model does not take into account caches.
3) They deal with simple index expressions of the form $c_1 * i + c_2$ and not a general affine function of the loop indices.

Our work complements the work of Wolf and Lam [3] and Schreiber and Dongarra [11]. Wolfe and Lam derive loop transformations (and tile the iteration space) to improve data locality in multiprocessors with caches. They use matrices to model transformations and use the notion of equivalence classes within the set of uniformly generated references to identify valid loop transformations to improve the degree of temporal and spatial locality within a given loop nest. Schreiber and Dongarra briefly address the problem of deriving optimal hyperparallelepiped iteration space tiles to minimize communication traffic (they refer to it as I/O requirements). However their work differs from this paper in the following ways:

1) Their machine model does not have a processor cache.
2) The data space corresponding to an array reference and the iteration space are isomorphic.

These restrictions make the problem of computing the communication traffic much simpler. Also, one of the main issues addressed by Schreiber and Dongarra is the *atomicity requirement* of the tiles which is related to the dependence vectors and this paper is not concerned with those requirements as it is assumed that the iterations can be executed in parallel.

Ferrante, Sarkar, and Thrash [12] address the problem of estimating the number of cache misses for a nest of loops. This problem is similar to our problem of finding the size of the cumulative footprint, but differs in these ways:

1) We consider a tile in the iteration space and not the entire iteration space; our tiles can be hyperparallelepipeds in general.
2) We partition the references into uniformly intersecting sets, which makes the problem computationally more tractable, since it allows us to deal with only the tile at the origin.
3) Our treatment of coupled subscripts is much simpler, since we look at maximal independent columns, as shown in Section V.B.

B. Overview of the Paper

The rest of this paper is structured as follows. Section II states our system model and our program-level assumptions. Section III first presents a few examples to illustrate the basic ideas behind loop partitioning; it then discusses the notion of data partitioning, and when it is important. Section IV develops the theoretical framework for partitioning and presents several additional examples. Section V extends the basic framework to handle more general expressions, and Section VI indicates modifications to the basic framework to handle data partitioning and more general types of systems. The framework for both loop and data partitioning has been implemented in the compiler system for the Alewife multiprocessor. The implementation of our compiler system and a sampling of results is presented in Section VII, and Section VIII concludes the paper.

II. Problem Domain and Assumptions

This paper focuses on the problem of partitioning loops in cache-coherent shared-memory multiprocessors. Partitioning involves deciding which loop iterations will run collectively in a thread of computation. Computing loop partitions involves finding the set of iterations which when run in parallel minimizes the volume of communication generated in the system. This section describes the types of programs currently handled by our framework and the structure of the system assumed by our analysis.

A. Program Assumptions

Fig. 1 shows the structure of the most general single loop nest that we consider in this paper. The statements in the *loop body* have array references of the form $A[\vec{g}(i_1, i_2, \ldots, i_l)]$, where the index function is $\vec{g} : Z^l \rightarrow Z^d$, l is the loop nesting and d is the dimension of the array A. We have restricted our attention to **doall** loops since we want to focus on the relation between the iteration space and the data space and factor out issues such as dependencies and synchronization that arise from the ordering of the iterations of a loop. We believe that the framework described in this paper can be applied with suitable modifications for loops in which the iterations are ordered.

```
Doall (i1=l1:u1, i2=l2:u2, ..., il=l1:ul)
  loop body
EndDoall
```

Fig. 1. Structure of a single loop nest.

We assume that all array references within the loop body are unconditional. One of the two following approaches may be taken for loops with conditionals.

- Assume that all array references are actually accessed, ignoring the conditions surrounding a reference.
- Include only references within conditions that are likely to be true based on profiling information.

We address the problem of loop and data partitioning for index expressions that are affine functions of loop indices. In other words, the index function can be expressed as,

$$\vec{g}(\vec{i}) = \vec{i}\,\mathbf{G} + \vec{a} \qquad (1)$$

where $\mathbf{G}$ is a $l \times d$ matrix with integer entries and $\vec{a}$ is an integer constant vector of length d, termed the *offset vector*. Note that $\vec{i}$, $\vec{g}(\vec{i})$, and $\vec{a}$ are row vectors. We often refer to an array reference by the pair $(\mathbf{G}, \vec{a})$. (An example of this function is presented in Section III.) Similar notation has been used in several papers in the past, for example, see [3], [4]. All our vectors and matrices have integer entries unless stated otherwise. We assume that the loop bounds are such that the iteration space is rectangular. The problem with nonrectangular tiles is one of load balancing (due to boundary effects in tiling) and this can be handled by optimizing for a machine with a large number of virtual processors and mapping the virtual processors to real processors in a cyclic fashion.

Loop indices are assumed to take all integer values between their lower and upper bounds, i.e, the strides are one.

Previous work [7], [8], [13] in this area restricted the arrays in the *loop body* to be of dimension exactly equal to the loop nesting. Abraham and Hudak [7] further restrict the *loop body* to contain only references to a single array; furthermore, all references are restricted to be of the form $A[i_1 + a_1, i_2 + a_2, \ldots, i_d + a_d]$ where a_j is an integer constant. Matrix multiplication is a simple example that does not fit these restrictions.

Given P processors, the problem of loop partitioning is to divide the iteration space into P tiles such that the total communication traffic on the network is minimized with the additional constraint that the tiles are of equal size, except at the boundaries of the iteration space. The constraint of equal size partitions is imposed to achieve load balancing. We restrict our discussions to hyperparallelepiped tiles, of which rectangular tiles are a special case.

Like [7], [8], [13], we do not include the effects of synchronization in our framework. Synchronization is handled separately to ensure correct behavior. For example, in the doall loop in Fig. 1, one might introduce a barrier synchronization after the loop nest if so desired. We also note that in many cases fine-grain data-level synchronization can be used within a parallel do loop to enforce data dependencies and its cost approximately modeled as slightly more expensive communication than usual [14]. See Appendix B for some details.

B. System Model

Our analysis applies to systems whose structure is similar to that shown in Fig. 2. The system comprises a set of processors, each with a coherent cache. Cache misses are satisfied by global memory accessed over an interconnection network or a bus. The memory can be implemented as a single monolithic module (as is commonly done in bus-based multiprocessors), or in a distributed fashion as shown in the figure. The memory modules might also be implemented on the processing nodes themselves (data partitioning for locality makes sense only for this case). In all cases, our analysis assumes that the cost of a main memory access is much higher than a cache access, and for loop partitioning, our analysis assumes that the cost of the main memory access is the same no matter where in main memory the data is located.

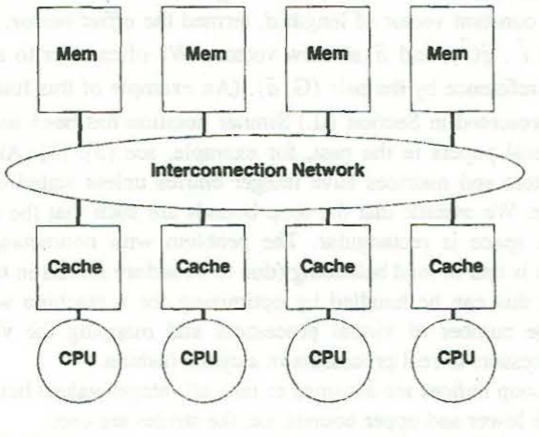


Fig. 2. A system with caches and uniform-access main memory (UMA).

The goal of loop partitioning is to minimize the total number of main memory accesses. For simplicity, we assume that the caches are large enough to hold all the data required by a loop partition, and that there are no conflicts in the caches. Techniques such as subblocking described in [15] or other techniques as in [17] and in [16] can be applied to reduce effects due to conflicts. When caches are small, the optimal loop partition does not change, rather, the size of each loop tile executed at any given time on the processor must be adjusted [15] so that the data fits in the cache (if we assume that the cache is effectively flushed between executions of each loop tile). Unless otherwise stated, we assume that cache lines are of unit length. The effect of larger cache lines can be included easily as suggested in [7], and is discussed further in Section VI.B.

If a program has multiple loops, then loop tiling parameters can be chosen independently for each loop to optimize cache performance by applying the techniques described in this paper. We assume there is no data reuse in the cache across loops. In programs with multiple loops and data arrays, tiling parameters for each loop and data array cannot be chosen independently in systems where the memories are local to the processors (see Fig. 5). This issue is discussed further in Section VI.C.

III. Loop Partitions and Data Partitions

This section presents examples to introduce and illustrate some of our definitions and to motivate the benefits of optimizing the shapes of loop and data tiles. More precise definitions are presented in the next section.

As mentioned previously, we deal with index expressions that are affine functions of loop indices. In other words, the index function can be expressed as in (1). Consider the following example to illustrate the above expression of index functions.

Example 1. The reference $A[i_3 + 2, 5, i_2 - 1, 4]$ in a triply nested loop can be expressed by

$$(i_1, i_2, i_3)\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 1 & 0 & 0 & 0\end{bmatrix} + (2, 5, -1, 4)$$

In this example, the second and fourth column of **G** are zero indicating that the second and fourth subscripts of the reference are independent of the loop indexes. In such cases, we show in Section V.B that we can ignore those columns and treat the referenced array as an array of lower dimension. In future, without loss of generality, we assume that the **G** matrix contains no zero columns.

Now, let us introduce the concept of a *loop partition* by examining the following example. Loop partitioning specifies the tiling parameters of the iteration space. Loop partitioning is sometimes termed iteration space partitioning or tiling.

Example 2.

```
Doall (i=101:200, j=1:100)
  A[i,j] = B[i+j,i-j-1]+B[i+j+4,i-j+3]
EndDoall
```

Let us assume that we have 100 processors and we want to distribute the work among them. There are 10,000 points in the iteration space and so one can allocate 100 of these to each of the processors to distribute the load uniformly. Fig. 3 shows two simple ways of partitioning the iteration space—by rows and by square blocks—into 100 equal tiles.

Minimizing communication volume requires that we minimize the number of data elements accessed by each loop tile. To facilitate this optimization, we introduce the notion of a *data footprint*. Footprints comprise the data elements referenced within a loop tile. In other words, the footprints are regions of the *data space* accessed by a loop tile. In particular, the footprint with respect to a specific reference in a loop tile gives us all the data elements accessed through that reference from within a tile of a loop partition.

Using Fig. 4, let us illustrate the footprints corresponding to a reference of the form `B[i+j,i-j-1]` for the two loop partitions shown in Fig. 3. The footprints in the data space resulting from the loop partition **a** are diagonal stripes and those resulting from partition **b** are square blocks rotated by 45 degrees. Algorithms for deriving the footprints are presented in the next section.

Let us compare the two loop partitions in the context of a system with caches and uniform-access memory (see Fig. 2) by computing the number of cache misses. The number of cache misses is equal to the number of distinct elements of `B` accessed by a loop tile, which is equal to the size of a loop tile's footprint on the array `B`. (Section VI.A deals with minimizing *cache-coherence* traffic). Caches automatically fetch a loop tile's data footprint as the loop tile executes. For each tile in partition **a**, the number of cache misses can be shown to be 104 (see Section V.A) whereas the number of cache misses in each tile of partition **b** can be shown to be 140. Thus, because it allows data reuse, loop partition **a** is a better choice if our goal is to minimize the number of cache misses, a fact that is not obvious from the source code.

When is *data partitioning* important? Data partitioning is the problem of partitioning the data arrays into data tiles and

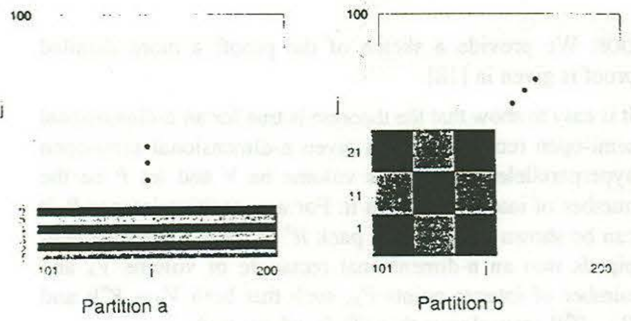
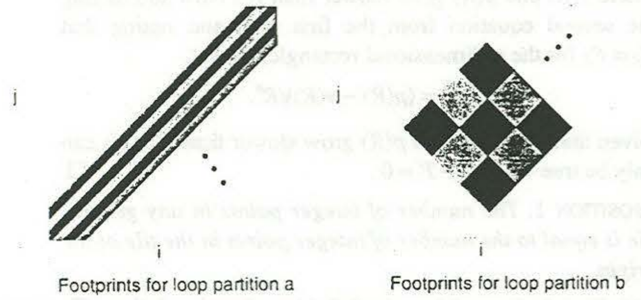


Fig. 3. Two simple rectangular loop partitions in the iteration space.



j
i
j
i
Footprints for loop partition a
Footprints for loop partition b

Fig. 4. Data footprints in the data space resulting from loop partitions a and b.

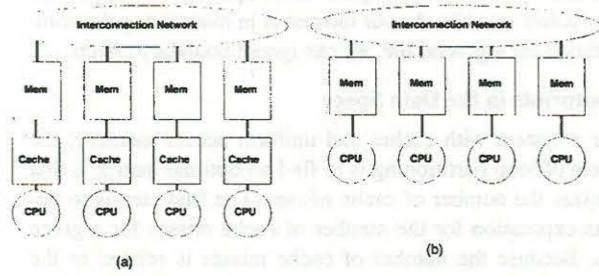


Fig. 5. Systems with nonuniform main-memory access time.

assigning each data tile to a local memory module, such that the number of memory references that can be satisfied by the local memory is maximized. Data partitioning is relevant only for nonuniform memory-access (NUMA) systems (for example, see Fig. 5).

In systems with nonuniform memory-access times, both loop and data partitioning are required. Our analysis applies to such systems as well. The loop tiles are assigned to the processing nodes and the data tiles to memory modules associated with the processing nodes so that a maximum number of the data references made by the loop tiles are satisfied by the local memory module. Note that in systems with nonuniform memory-access times, but which have caches, data partitioning may still be performed to maximize the number of caches misses that can be satisfied by the memory module local to the processing node.

Referring to Fig. 4, the footprint size is minimized by choosing a diagonal striping of the data space as the data partition, and a corresponding horizontal striping of the iteration space as the loop partition. The additional step of aligning corresponding loop and data tiles on the same node maximizes the number of local memory references.

In fact, the above horizontal partitioning of the loop space and diagonal striping of the data space results in zero communication traffic. Ramanujam and Sadayappan [8] presented algorithms to derive such communication-free partitions when possible. On the other hand, in addition to producing the same partitions when communication-traffic-free partitions exist (see Sections V.A and VI.C), our analysis will discover partitions that minimize traffic when such partitions are non-existent as well (see Example 8).

EXAMPLE 3.

```
Doall (i=1:N, j=1:N)
  A[i,j]=B[i,j]+B[i+1,j-2]+B[i-1,j+1]
EndDoall
```

For the loop shown in Example 3, a parallelogram partition results in a lower cost of memory access compared to any rectangular partition since most of the inter iteration communication can be internalized to within a processor for a parallelogram partition (see Section VII.A). Because rectangular partitions often do not minimize communication, we would like to include parallelograms in the formulation of the general loop partitioning problem. In higher dimensions a parallelogram tile generalizes to a hyperparallelepiped; the next section defines it precisely.

IV. A Framework for Loop and Data Partitioning

This section first defines precisely the notion of a loop partition and the notion of a footprint of a loop partition with respect to a data reference in the loop. We prove a theorem showing that the number of integer points within a tile is equal to the volume of the tile, which allows us to use volume estimates in deriving the amount of communication. We then present the concept of uniformly intersecting references and a method of computing the cumulative footprint for a set of uniformly intersecting references. We develop a formalism for computing the volume of communication on the interconnection network of a multiprocessor for a given loop partition, and show how loop tiles can be chosen to minimize this traffic. We briefly indicate how the cumulative footprint can be used to derive optimal data partitions for multicomputers with local memory (NUMA machines).

A. Loop Tiles in the Iteration Space

Loop partitioning results in a tiling of the iteration space. We consider only hyperparallelepiped partitions in this paper; rectangular partitions are special cases of these. Furthermore, we focus on loop partitioning where the tiles are homogeneous except at the boundaries of the iteration space. Under these conditions of homogeneous tiling, the partitioning is completely defined by specifying the tile at the origin, as indicated in Fig. 6. Under homogeneous tiling, the concept of the tile at the origin is similar to the notion of the clustering basis in [6]. (See Appendix A for a more general representation of hyperparallelepiped loop tiles based on bounding hyperplanes.)

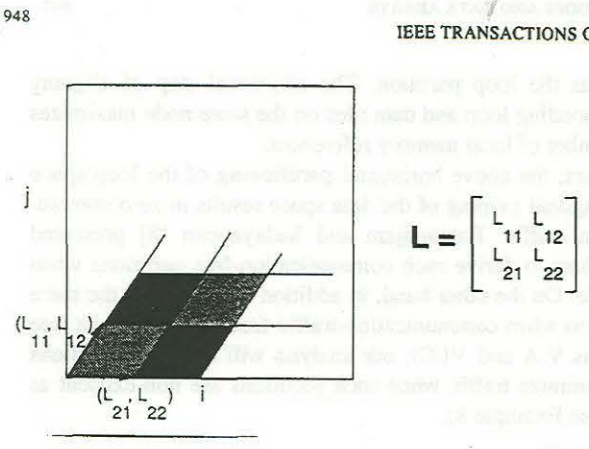

j
$\mathbf{L} = \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix}$
(L_{11}, L_{12})
(L_{21}, L_{22})
i

Fig. 6. Iteration space partitioning is completely specified by the tile at the origin.

DEFINITION 1. *An l dimensional square integer matrix* $\mathbf{L}$ *defines a semiopen hyperparallelepiped tile at the origin of an l dimensional iteration space as follows. The set of iteration points included in the tile is*

$$\left\{ \vec{x} \,\middle|\, \vec{x} = \sum_{i=1}^{l} \alpha_i \vec{l}_i, 0 \le \alpha_i < 1 \right\}$$

where $\vec{l}_i$ *is the ith row of* $\mathbf{L}$. *As depicted in Fig. 6, the rows of the matrix* $\mathbf{L}$ *specify the vertices of the tile at the origin. Often, we also refer to the partition by the* $\mathbf{L}$ *matrix since each of the other tiles is a translation of the tile at the origin.*

EXAMPLE 4. A rectangular partition can be represented by a diagonal $\mathbf{L}$ matrix. Consider a three dimensional iteration space $I \times J \times K$ partitioned into rectangular tiles where each tile is of the form by $\{(i_0, j, k_0) \mid 0 \le j < J\}$. In other words, constants i_0 and j_0 specify the tile completely. Such a partition is represented by

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & J & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

DEFINITION 2. *A general tile in the iteration space is a translation of the tile at the origin. The translation vector is given by*

$$\sum_{i=1}^{l} \lambda_i \vec{l}_i$$

where λ_i *is an integer. A tile is completely specified by* $(\lambda_1, \ldots, \lambda_l)$. *For example* $(0, \ldots, 0)$ *specifies the tile at the origin.*

The rest of this paper deals with optimizing the shape of the tile at the origin for minimal communication. Because the amount of communication is related to the number of integer points within a tile, we begin by proving the following theorem relating the volume of a tile to the number of integer points within it. This theorem on lattices allows us to use volumes of hyperparallelepipeds derived using determinants to determine the amount of communication.

THEOREM 1. *The number of integer points (iteration points) in tile* $\mathbf{L}$ *is equal to the volume of the tile, which is given by* $|\det \mathbf{L}|$.

PROOF. We provide a sketch of the proof; a more detailed proof is given in [18].

It is easy to show that the theorem is true for an n-dimensional semi-open rectangle. For a given n-dimensional semi-open hyperparallelepiped, let its volume be V and let P be the number of integer points in it. For any positive integer R, it can be shown that one can pack R^n of these hyperparallelepipeds into an n-dimensional rectangle of volume V_R and number of integer points P_R, such that both $V_R - R^n V$ and $P_R - R^n P$ grow slower than R^n. In other words,

$$V_R = R^n V + v(R), \;\; P_R = R^n P + p(R)$$

where $v(R)$ and $p(R)$ grow slower than R^n. Now subtracting the second equation from the first one, and noting that $V_R = P_R$ for the n-dimensional rectangle, we get,

$$V - P = (p(R) - v(R))/R^n.$$

Given that both $v(R)$ and $p(R)$ grow slower than R^n, this can only be true when $V - P = 0$. □

PROPOSITION 1. *The number of integer points in any general tile is equal to the number of integer points in the tile at the origin.*

PROOF. Straight-forward from the definition of a general tile. □

In the following discussion, we ignore the effects of the boundaries of the iteration space in computing the number of integer points in a tile. As our interest is in minimizing the communication for a general tile, we can ignore boundary effects.

B. Footprints in the Data Space

For a system with caches and uniform access memory, the problem of loop partitioning is to find an optimal matrix $\mathbf{L}$ that minimizes the number of cache misses. The first step is to derive an expression for the number of cache misses for a given tile $\mathbf{L}$. Because the number of cache misses is related to the number of unique data elements accessed, we introduce the notion of a *footprint* that defines the data elements accessed by a tile. The footprints are regions of the *data space* accessed by a loop tile.

DEFINITION 3. *The* **footprint** *of a tile of a loop partition with respect to a reference* $A[\vec{g}(\vec{i})]$ *is the set of all data elements* $A[\vec{g}(\vec{i})]$ *of A, for* $\vec{i}$ *an element of the tile.*

The footprint gives us all the data elements accessed through a particular reference from within a tile of a loop partition. Because we consider homogeneous loop tiles, the number of data elements accessed is the same for each loop tile.

We will compute the number of cache misses for the system with caches and uniform access memory to illustrate the use of footprints. The body of the loop may contain references to several variables and we assume that aliasing has been resolved; two references with distinct names do not refer to the same location. Let $A_1, A_2, \ldots, A_K$ be references to array A within the loop body, and let $f(A_i)$ be the *footprint* of the loop tile at the origin with respect to the reference A_i and let

$$f(A_1, A_2, \ldots, A_K) = \bigcup_{i=1,\ldots K} f(A_i)$$

be the *cumulative footprint* of the tile at the origin. The number of cache misses with respect to the array A is $|f(A_1, A_2, \ldots, A_K)|$. Thus, computing the size of the individual footprints and the size of their union is an important part of the loop partitioning problem.

To facilitate computing the size of the union of the footprints we divide the references into multiple disjoint sets. If two footprints are disjoint or mostly disjoint, then the corresponding references are placed in different sets, and the size of the union is simply the sum of the sizes of the two footprints.

However, references whose footprints overlap substantially are placed in the same set. The notion of *uniformly intersecting references* is introduced to specify precisely the idea of "substantial overlap." Overlap produces temporal locality in cache accesses, and computing the size of the union of their footprints is more complicated.

The notion of uniformly intersecting references is derived from definitions of intersecting references and uniformly generated references.

DEFINITION 4. *Two references $A[\vec{g}_1(\vec{i})]$ and $A[\vec{g}_2(\vec{i})]$ are said to be intersecting if there are two integer vectors $\vec{i}_1$, $\vec{i}_2$ such that $\vec{g}_1(\vec{i}_1) = \vec{g}_2(\vec{i}_2)$. For example, $A[i + c1, j + c2]$ and $A[j + c3, i + c4]$ are intersecting, whereas $A[2i]$ and $A[2i + 1]$ are nonintersecting.*

DEFINITION 5. *Two references $A[\vec{g}_1(\vec{i})]$ and $A[\vec{g}_2(\vec{i})]$ are said to be* uniformly generated *if*

$$g_1(\vec{i}) = \vec{i}\mathbf{G} + \vec{a}_1 \quad \text{and} \quad g_2(\vec{i}) = \vec{i}\mathbf{G} + \vec{a}_2$$

where $\mathbf{G}$ *is a linear transformation and $\vec{a}_1$ and $\vec{a}_2$ are integer constants.*

The intersection of footprints of two references that are not uniformly generated is often very small. For nonuniformly generated references, although the footprints corresponding to some of the iteration-space tiles might overlap partially, the footprints of others will have no overlap. Since we are interested in the worst-case communication volume between any pair of footprints, we will assume that the total communication generated by two nonuniformly intersecting references is essentially the sum of the individual footprints.

However, the condition that two references are uniformly generated is not sufficient for two references to be intersecting. As a simple example, $A[2i]$ and $A[2i + 1]$ are uniformly generated, but the footprints of the two references do not intersect. For the purpose of locality optimization through loop partitioning, our definition of reuse of array references will combine the concept of uniformly generated arrays and the notion of intersecting array references. This notion is similar to the equivalence classes within uniformly generated references defined in [3].

DEFINITION 6. *Two array references are* uniformly intersecting *if they are both intersecting and uniformly generated.*

EXAMPLE 5. The following sets of references are uniformly intersecting.

1) $A[i, j], A[i + 1, j - 3], A[i, j + 4]$.

2) $A[2j, 2, i], A[2j - 5, 2, i], A[2j + 3, 2, i]$.

The following pairs are not uniformly intersecting.

1) $A[i, j], A[2i, j]$.
2) $A[i, j], A[2i, 2j]$.
3) $A[j, 2, i], A[j, 3, i]$.
4) $A[2i], A[2i + 1]$.
5) $A[i + 2, 2i + 4], A[i + 5, 2i + 8]$.
6) $A[i, j], B[i, j]$.

Footprints in the data space for a set of uniformly intersecting references are translations of one another, as shown below. The footprint with respect to the reference $(\mathbf{G}, \vec{a}_s)$ is a translation of the footprint with respect to the reference $(\mathbf{G}, \vec{a}_r)$, where the translation vector is $\vec{a}_s - \vec{a}_r$.

PROPOSITION 2. *Given a loop tile at the origin* $\mathbf{L}$ *and references $r = (\mathbf{G}, \vec{a}_r)$ and $s = (\mathbf{G}, \vec{a}_s)$ belonging to a uniformly generated set defined by* $\mathbf{G}$, *let $f(r)$ denote the footprint of* $\mathbf{L}$ *with respect to r, and let $f(s)$ denote the footprint of* $\mathbf{L}$ *with respect to s. Then $f(s)$ is simply a translation of $f(r)$, where each point of $f(s)$ is a translation of a corresponding point of $f(r)$ by an amount given by the vector $(\vec{a}_s - \vec{a}_r)$. In other words,*

$$f(s) = f(r) + (\vec{a}_s - \vec{a}_r)$$

This follows directly from the definition of uniformly generated references. Recall that an element $\vec{i}$ of the loop tile is mapped by the reference $(\mathbf{G}, \vec{a}_r)$ to data element $\vec{d}_r = \vec{i}\mathbf{G} + \vec{a}_r$, and by the reference $(\mathbf{G}, \vec{a}_s)$ to data element $\vec{d}_s = \vec{i}\mathbf{G} + \vec{a}_s$. The translation vector, $(\vec{d}_s - \vec{d}_r)$, is clearly independent of $\vec{i}$.

The volume of cache traffic imposed on the network is related to the size of the cumulative footprint. We describe how to compute the size of the cumulative footprint in the following two sections as outlined below.

- First, we discuss how the size of *the footprint for a single reference* within a loop tile can be computed. In general, the size of the footprint with respect to a given reference is not the same as the number of points in the iteration space tile.
- Second, we describe how the size of *the cumulative footprint for a set of uniformly intersecting references* can be computed. The sizes of the cumulative footprints for each of these sets are then summed to produce the size of the cumulative footprint for the loop tile.

C. Size of a Footprint for a Single Reference

This section shows how to compute the size of the footprint (with respect to a given reference and a given loop tile $\mathbf{L}$) efficiently for certain common cases of $\mathbf{G}$. The general case of $\mathbf{G}$ is dealt with in Section V. We begin with a simple example to illustrate our approach.

EXAMPLE 6

```
Doall (i=0:99, j=0:99)
  A[i,j]=B[i+j,j]+B[i+j+1,j+2]
EndDoall
```

The reference matrix $\mathbf{G}$ is

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Let us suppose that the loop tile at the origin L is given by

$$\begin{bmatrix} L_1 & L_1 \\ L_2 & 0 \end{bmatrix}$$

Fig. 7 shows this tile at the origin of the iteration space and the footprint of the tile (at the origin) with respect to the reference $B[i+j, j]$ is shown in Fig. 8. The matrix

$$\mathbf{f}(B[i+j,\ j]) = \mathbf{LG} = \begin{bmatrix} 2L_1 & L_1 \\ L_2 & 0 \end{bmatrix}$$

describes the footprint. As shown later, the integer points in the semi open parallelogram specified by **LG** is the footprint of the tile and so the size of the footprint is |det(**LG**)|. We will use **D** to denote the product **LG** as it appears often in our discussion.

The rest of this subsection focuses on deriving the set of conditions under which the footprint size is given by |det(**D**)|. Briefly, we show that **G** being unimodular is a sufficient (but not necessary) condition. The next section derives the size of the cumulative footprint for multiple uniformly intersecting references.

In general, is the footprint exactly the integer points in **D** = **LG**? If not, how do we compute the footprint? The first question can be expanded into the following two questions.

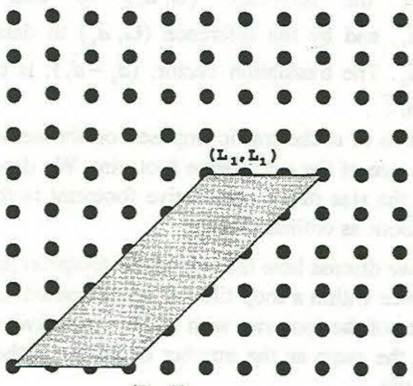


Fig. 7. Tile **L** at the origin of the iteration space.

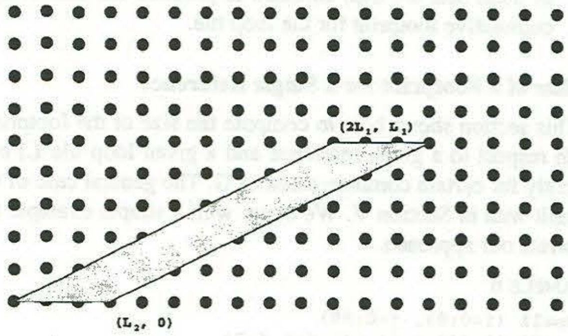


Fig. 8. Footprint of **L** wrt $B[i+j, j]$ in the data space.

- Is there a point in the footprint that lies outside the hyperparallelepiped **D**? It follows easily from linear algebra that it is not the case.
- Is every integer point in **D** an element of the footprint? It is easy to show this is not true and a simple example corresponds to the reference $A[2i]$.

We first study the simple case when the hyperparallelepiped **D** completely defines the footprint. A precise definition of the set $S(\mathbf{D})$ of points defined by the matrix **D** is as follows.

DEFINITION 7. *Given a matrix* **D** *whose rows are the vectors* $\vec{d}_i, 1 \le i \le m$, *$S(\mathbf{D})$ is defined as the set*

$$\left\{ \vec{x} \,\middle|\, \vec{x} = a_1\vec{d}_1 + a_2\vec{d}_2 + \ldots + a_m\vec{d}_m,\ 0 \le a_i < 1 \right\}.$$

$S(\mathbf{D})$ defines all the points in the semi open hyperparallelepiped defined by **D**.

So for the case where **D** completely defines the footprint, the footprint is exactly the integer points in $S(\mathbf{D})$. One of the cases where **D** completely defines the footprint, is when **G** is unimodular as shown below.

LEMMA 1. *The mapping $\Re^l \mapsto \Re^d$ as defined by* **G** *is one to one if and only if the rows of* **G** *are independent. Further, the mapping of the iteration space to the data space ($Z^l \mapsto Z^d$) as defined by* **G** *is one to one if and only if the rows of* **G** *are independent.*

PROOF. $\vec{i}_1\mathbf{G} = \vec{i}_2\mathbf{G}$ implies $\vec{i}_1 = \vec{i}_2$ if and only if the only solution to $\vec{x}\mathbf{G} = \vec{0}$ is $\vec{0}$. The latter implies that the nullspace of $\mathbf{G}^T$ is of dimension 0. From a fundamental theorem of linear algebra [19], this means that the rows of **G** are linearly independent. It is to be noted that when the rows of **G** are not independent there exists a nontrivial integer solution to $\vec{x}\mathbf{G} = \vec{0}$, given that the entries in **G** are integers. This proves the second statement of the lemma. □

LEMMA 2. *The mapping of the iteration space to the data space as defined by* **G** *is onto if and only if the columns of* **G** *are independent and the g.c.d. of the subdeterminants of order equal to the number of columns is* 1.

PROOF. Follows from the Hermite normal form theorem as shown in [20]. □

LEMMA 3. *If* **G** *is invertible then $\vec{d} \in \mathbf{LG}$ if and only if $\vec{d}\mathbf{G}^{-1} \in \mathbf{L}$.*

PROOF. Clearly **G** is invertible implies,

$$\vec{d} \in \mathbf{LG} \Rightarrow \vec{d}\mathbf{G}^{-1} \in \mathbf{LGG}^{-1} = \mathbf{L}$$

Also,

$$\vec{d}\mathbf{G}^{-1} \in \mathbf{L} \Rightarrow \vec{d}\mathbf{G}^{-1}\mathbf{G} \in \mathbf{LG} \Rightarrow \vec{d} \in \mathbf{LG}.$$

G is invertible implies that the rows of **G** are independent and hence the mapping defined by **G** is one to one from Lemma 1. □

THEOREM 2. *The footprint of the tile defined by* **L** *with respect to the reference* **G** *is identical to the integer points in the semi open hyperparallelepiped* **D** = **LG** *if* **G** *is* unimodular.

PROOF. It is immediate from Lemma 2 that **G** is onto when it is unimodular. **G** is onto implies that every data point in **D** has an inverse in the iteration space. Can the inverse of the data point be outside of **L**? Lemma 3 shows this is not possible since **G** is invertible. □

We make the following two observations about Theorem 2.

- **G** is unimodular is a sufficient condition; but not necessary. An example corresponds to the reference $A[i + j]$. Further discussions on this is contained in Section V.
- One may wonder why **G** being onto is not sufficient for **D** to coincide with the footprint. Even when every integer point in **D** has an inverse, it is possible that the inverse is outside of **L**. For example, consider the mapping defined by the **G** matrix

$$\begin{bmatrix} 4 \\ 5 \end{bmatrix}$$

corresponding to the reference $A[4i + 5j]$. It is onto as shown by Lemma 2. However, we will show that not all points in **LG** are in the footprint. Consider,

$$\mathbf{L} = \begin{bmatrix} 100 & 0 \\ 0 & 100 \end{bmatrix}.$$

LG defines the interval [0, 900) and so it includes the data point (1). But it can be shown that none of the inverses of the data point (1) belong to **L**; (–1, 1) is an inverse of (1). The same is true for the data points (2),(3),(6),(7), and (11). The one to one property of **G** guarantees that no point from outside of **L** can be mapped to inside of **D**. The reason for this is that the one to one property is true even when **G** is treated as a function on reals.

Let us now introduce our technique for computing the cumulative footprint when **G** is unimodular. Algorithms for computing the size of the individual footprints and the cumulative footprint when **G** is not unimodular are discussed in Section V.

D. Size of the Cumulative Footprint

The size of the cumulative footprint F for a loop tile is computed by summing the sizes of the cumulative footprints for each of the sets of uniformly intersecting references. This section presents a method for computing the size of the cumulative footprint for a set of uniformly intersecting references when **G** is unimodular, that is, when the conditions stated in Theorem 2 are true. More general cases of **G** are discussed in Section V. We first describe the method when there are exactly two uniformly intersecting references, and then develop the method for multiple references.

Cumulative Footprint for Two References. Let us start by illustrating the computation of the cumulative footprint for Example 6. The two references to array B form a uniformly intersecting set and are defined by the following **G** matrix.

$$\mathbf{G} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Let us suppose that the loop partition **L** is given by

$$\begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix}.$$

Then **D** is given by

$$\begin{bmatrix} L_{11} + L_{12} & L_{12} \\ L_{21} + L_{22} & L_{22} \end{bmatrix}.$$

The parallelogram defined by **D** in the data space is the parallelogram $ABCD$ shown in Fig. 9. $ABCD$ and $EFGH$ shown in Fig. 9 are the footprints of the tile **L** with respect to the two references ($B[i + j, j]$ and $B[i + j + 1, j + 2]$, respectively) to array B. In the figure, $\vec{AB} = (L_{11} + L_{12}, L_{12})$, $\vec{AD} = (L_{21} + L_{22}, L_{22})$, and $\vec{AE} = (1, 2)$.

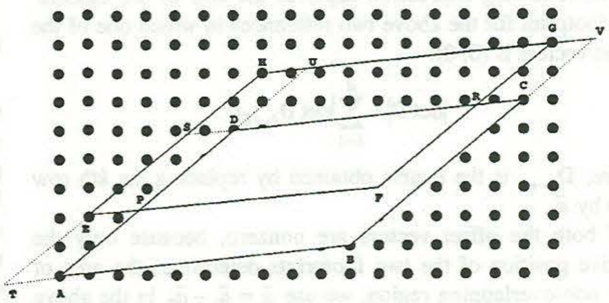


Fig. 9. Data footprint wrt $B[i + j, j]$ and $B[i + j + 1, j + 2]$.

The size of the cumulative footprint is the size of footprint $ABCD$ plus the number of data elements in $EPDS$ plus the number of data elements in $SRGH$. Given that **G** is unimodular, the number of data elements is equal to the area $ABCD$ + $SRGH$ + $EPDS$ = $ABCD$ + $ADST$ + $CDUV$ – $SDUH$. Ignoring the area $SDUH$, we can approximate the total area by

$$\left|\det\begin{bmatrix} L_{11} + L_{12} & L_{12} \\ L_{21} + L_{22} & L_{22} \end{bmatrix}\right| + \left|\det\begin{bmatrix} L_{11} + L_{12} & L_{12} \\ 1 & 2 \end{bmatrix}\right| + \left|\det\begin{bmatrix} 1 & 2 \\ L_{21} + L_{22} & L_{22} \end{bmatrix}\right|.$$

The first term in the above equation represents the area of the footprint of a single reference, i.e., |det(**D**)|. It is well known that the area of a parallelogram is given by the determinant of the matrix specifying the parallelogram. The second and third terms are the determinants of the **D** matrix in which one row is replaced by the offset vector $\bar{a} = (1, 2)$. Fig. 10 is a pictorial representation of the approximation. The first term is the parallelogram $ABCD$ and the second and third terms are the shaded regions.

Ignoring $SDUH$ is reasonable if we assume that the offset vectors in a uniformly intersecting set of references are small compared to the tile size. We refer to this simplification as the *overlapping subtile approximation*. This approximation will result in our estimates being higher than the actual values. Although one can easily derive a more exact expression, we use the overlapping subtile approximation to simplify the computation. Fig. 16 in Section VII further demonstrates that the error introduced is insignificant, especially for parallelograms that are near optimal.

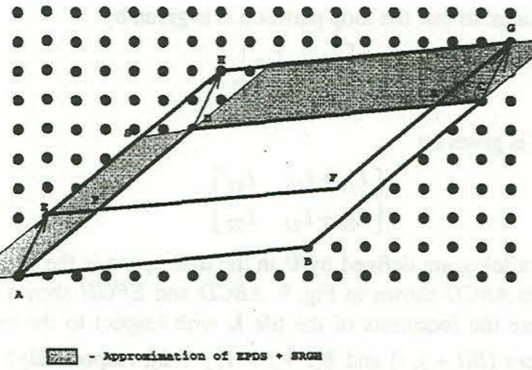


Fig. 10. Difference between the cumulative footprint and the footprint.

The following expression captures the size of the cumulative footprint for the above two references in which one of the offset vectors is (0, 0):

$$|\det \mathbf{D}| + \sum_{k=1}^{d} |\det \mathbf{D}_{k\to\vec{a}}|$$

where, $\mathbf{D}_{k\to\vec{a}}$ is the matrix obtained by replacing the *k*th row of **D** by $\vec{a}$.

If both the offset vectors are nonzero, because only the relative position of the two footprints determines the area of their non-overlapping region, we use $\vec{a} = \vec{a}_1 - \vec{a}_0$ in the above equation. The following discussion formalizes this notion and extends it to multiple references.

Cumulative Footprint for Multiple References. The basic approach for estimating the cumulative footprint size involves deriving an effective offset vector $\hat{a}$ that captures the combined effects of multiple offset vectors when there are several overlapping footprints resulting from a set of uniformly intersecting references. First, we need a few definitions.

DEFINITION 8. *Given a loop tile* **L**, *there are two neighboring loop tiles along the* i*th row of* **L** *defined by* $\{\vec{y} \mid \vec{y} = \vec{x} + \vec{l}_i, \vec{x} \in \text{tile } \mathbf{L}\}$ *and* $\{\vec{y} \mid \vec{y} = \vec{x} - \vec{l}_i, \vec{x} \in \text{tile } \mathbf{L}\}$, *where* $\vec{l}_i$ *is the* i*th row of* **L**, *for* $1 \le i \le l$. *We refer to the former neighbor as the positive neighbor and the latter as the negative neighbor. We also refer to these neighbors as the neighbors of the parallel sides of the tile determined by the rows of* **L**, *excluding the* i*th row. Fig. 11 illustrates the notion of neighboring tiles.*

The notion of neighboring tiles can be extended to the data space in like manner as follows.

DEFINITION 9. *Given a loop tile* **L** *and a reference* $(\mathbf{G}, \vec{a}_r)$, *the neighbors of the data footprint of* **L** *along the* k*th row of* $\mathbf{D} = \mathbf{LG}$ *are* $\{\vec{y} \mid \vec{y} = \vec{x} + \vec{d}_k, \vec{x} \in \mathbf{D} + \vec{a}_r\}$ *and* $\{\vec{y} \mid \vec{y} = \vec{x} - \vec{d}_k, \vec{x} \in \mathbf{D} + \vec{a}_r\}$, *where* $\vec{d}_k$ *is the* k*th row of* **D**, *for* $1 \le k \le d$.

DEFINITION 10. *Given a tile* **L**, **L′** *is a subtile wrt the* i*th row of* **L** *if the rows of* **L′** *are the same as the rows of* **L** *except for the* i*th row which is* α *times the* i*th row of* **L**, $0 \le \alpha \le 1$.

The approximation of the cumulative footprint in Fig. 10 can be expressed in terms of subtiles of the tile in the data

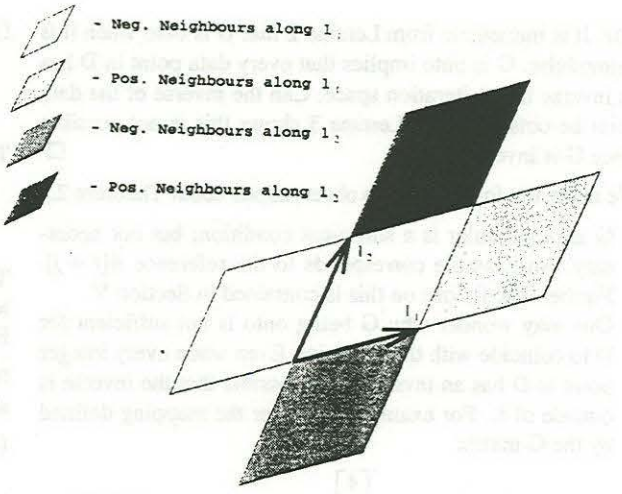

- Neg. Neighbours along l_1
- Pos. Neighbours along l_1
- Neg. Neighbours along l_2
- Pos. Neighbours along l_2
l_2
l_1

Fig. 11. Neighboring tiles.

space. *ABCD* is a tile in the data space and the two shaded regions in Fig. 10 are subtiles of neighboring tiles containing portions of the cumulative footprint. One can view the cumulative footprint as any one of the footprints together with communication from the neighboring footprints. The approximation of the cumulative footprint expresses the communication from the neighboring tiles in terms of subtiles to make the computation simpler.

DEFINITION 11. *Let* **L** *be a loop tile at the origin, and let* $\vec{g}(\vec{i}) = \vec{i}\mathbf{G} + \vec{a}_r$, $1 \le r \le R$ *be a set of uniformly intersecting references. For the footprint of* **L** *with respect to reference* $(\mathbf{G}, \vec{a}_r)$, *communication along the positive direction of the* k*th row of* **D** *is defined as the smallest sub tile of the positive neighbor along the* k*th row of the footprint which contains the elements of the cumulative footprint within that neighbor. Communication along the negative direction is defined similarly. Communication along the* k*th row is the sum of these two communications. Each row of* **D** *defines a pair of parallel sides (hyperplanes) of the data footprint determined by the remaining rows of* **D**. *We sometimes refer to the communication along the* k*th row as the communication across the parallel sides of* **D** *defined by the* k*th row.*

The notion of the communication along the rows of **D** facilitates computing the size of the cumulative footprint. Consider the data footprints of a loop tile with respect to a set of uniformly intersecting references shown in Fig. 12. Here $\vec{d}_1$, $\vec{d}_2$ correspond to the rows of the matrix $\mathbf{D} = \mathbf{LG}$. The vectors $\vec{a}_1, \ldots, \vec{a}_5$, are the offset vectors corresponding to the set of uniformly intersecting references. The cumulative footprint can be expressed as the union of any one of the footprints and the remaining elements of the cumulative footprint. We take the union because a given data element needs to be fetched only once into a cache.

In Fig. 12, the cumulative footprint is the union of the footprint of the loop tile with respect to $\vec{a}_4$ and the shaded regions corresponding to the remaining elements of the cumulative

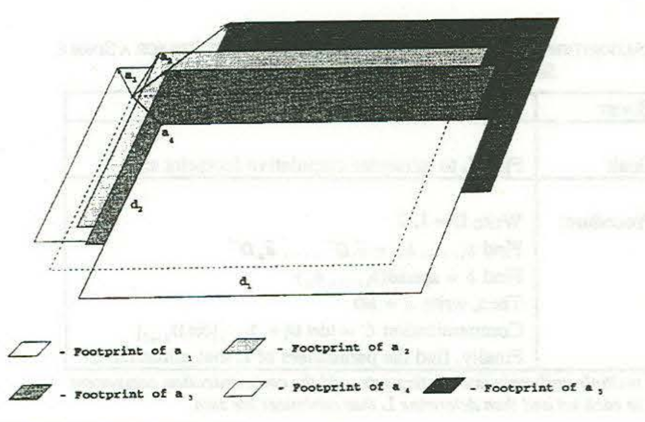


Fig. 12. Cumulative footprint.

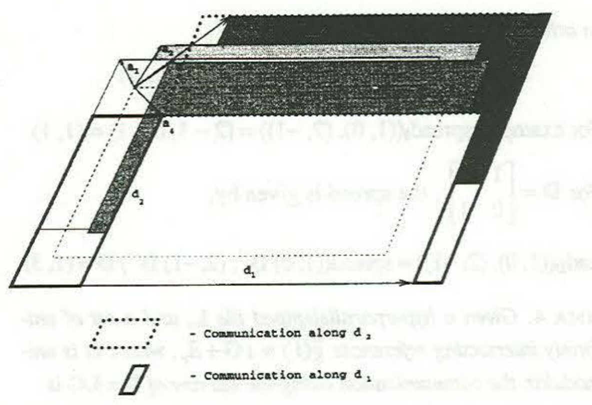


Fig. 13. Communication from neighboring tiles.

footprint resulting from the other references. The area of the shaded region can be approximated by the sum of communication along the kth row for $1 \le k \le 2$ as shown in Fig. 13. The area of the communication along $\vec{d}_2$ is equal to the area of the parallelogram whose sides are $\vec{d}_1$ and $\vec{a}_5 - \vec{a}_4$. Among the offset vectors, vector $\vec{a}_5$ has the maximum component along $\vec{d}_2$ and vector $\vec{a}_4$ has the minimum (taking the sign into account) component along $\vec{d}_2$. Similarly the area of the communication along $\vec{d}_1$ is equal to the area of the parallelogram whose sides are $\vec{d}_2$ and $\vec{a}_4 - \vec{a}_1$ plus the area of the parallelogram whose sides are $\vec{d}_2$ and $\vec{a}_5 - \vec{a}_4$. This is equal to the area of the parallelogram whose sides are $\vec{d}_2$ and $\vec{a}_5 - \vec{a}_1$. As before among the offset vectors, vector $\vec{a}_5$ has the maximum component along $\vec{d}_1$ and vector $\vec{a}_1$ has the minimum (taking the sign into account) component along $\vec{d}_1$. This observation is used in the proof of Theorem 3. It turns out that the effect of offset vector $\vec{a}_5 - \vec{a}_1$ along $\vec{d}_2$ and $\vec{a}_5 - \vec{a}_4$ along $\vec{d}_1$ can be captured by a single vector $\hat{a}$ as shown later.

PROPOSITION 3. *Let* **L** *be a loop tile at the origin, and let* $\vec{g}(\vec{i}) = \vec{i}\mathbf{G} + \vec{a}_r,\}$ *be a set of uniformly intersecting references. The volume of communication along the* k*th row of* **D**, $1 \le k \le d$, *is the same for each of the footprints (corresponding to the different offset vectors).*

Communication along the positive and negative directions will be different for different footprints. But the total communication along the kth row, $1 \le k \le d$, is the same for each of the data footprints.

We now derive an expression for the cumulative footprint based on our notion of communication across the sides of the data footprint. Our goal is to capture in a single offset vector $\hat{a}$ the communication in a cache-coherent system resulting from all the offset vectors. More specifically, we would like the kth component of $\hat{a}$ to reflect the communication per unit area across the parallel sides defined by the kth row of **D**. The effective vector $\hat{a}$ is derived from the *spread* of a set of offset vectors.

DEFINITION 12. *Given a set of d-dimensional offset vectors* $\vec{a}_r$, $1 \le r \le R$, *spread* $(\vec{a}_1, \ldots, \vec{a}_R)$ *is a vector of the same dimension as the offset vectors, whose* k*th component is given by*

$$\max_r(a_{r,k}) - \min_r(a_{r,k}),\ \forall k \in 1, \ldots, d.$$

In other words, the spread of a set of vectors is a vector in which each component is the difference between the maximum and minimum of the corresponding components in each of the vectors.

For caches, we use the *max − min* formulation (or the spread) to calculate the amount of communication traffic because the data space points corresponding to the footprints whose offset vectors have values between the *max* and the *min* lie within the cumulative footprint calculated using the spread.[2]

The spread as defined above does not quite capture the properties that we are looking for in a single offset vector except when **D** is rectangular. If **D** is not rectangular, the kth component of spread $(\vec{a})$ does not reflect the communication per unit area across the parallel sides defined by the kth row of **D**. To derive the footprint component (or subtile) along a row of **D**, we need to compute the difference between the maximum and the minimum components of the offset vectors using **D** as a basis. Therefore, we extend the notion of spread to a general basis as follows. Recall that **D** is a basis for the data space when **G** is unimodular.

In the definition below, $\vec{b}_r$ is the representation of offset vector $\vec{a}_r$ using **D** as the basis.

DEFINITION 13. *Given a set of offset vectors* $\vec{a}_r$, $1 \le r \le R$, *let* $\vec{b}_r = \vec{a}_r\mathbf{D}^{-1}, \forall r \in 1, \ldots, R$ *and let* $\hat{b}$ *be spread*$(\vec{b}_1, \ldots, \vec{b}_R)$. *Then*

$$\hat{a} = \text{spread}_{\mathbf{D}}(\vec{a}_1, \ldots, \vec{a}_R) = \hat{b}\mathbf{D}.$$

Looking at the special case where **D** is rectangular helps in understanding the definition.

PROPOSITION 4. *If* **D** *is rectangular then*

$$\hat{a} = \text{spread}(\vec{a}_1, \ldots, \vec{a}_R) = \text{spread}_{\mathbf{D}}(\vec{a}_1, \ldots, \vec{a}_R)$$

2. For data partitioning, however, the formulation must be modified as discussed in Section VI.C.

In other words,

$$\hat{a}_k = \max_r(a_{r,k}) - \min_r(a_{r,k}), \quad \forall k \in 1, \ldots d.$$

For example, $\text{spread}_{\mathbf{I}}((1, 0), (2, -1)) = (2 - 1, 0 - 1) = (1, 1)$.

For $\mathbf{D} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, the spread is given by,

$\text{spread}_{\mathbf{D}}((1, 0), (2, -1)) = \text{spread}((1, 0)\,\mathbf{D}^{-1}, (2, -1)\,\mathbf{D}^{-1})\,\mathbf{D} = (1, 3)$

LEMMA 4. *Given a hyperparallelepiped tile* **L**, *and a set of uniformly intersecting references* $\vec{g}(\vec{i}) = \vec{i}\,\mathbf{G} + \vec{a}_r$, *where* **G** *is unimodular, the communication along the* k*th row of* **D** = **LG** *is*

$$\sum_{k=1}^{d} \left|\det \mathbf{D}_{k \to \hat{a}}\right|$$

where $\hat{a} = \text{spread}_{\mathbf{D}}(\vec{a}_1, \ldots, \vec{a}_R)$ *and* $\mathbf{D}_{k \to \hat{a}}$ *is the matrix obtained by replacing the* k*th row of* **D** *by* $\hat{a}$.

PROOF. Straight-forward from the definition of spread and the definition of communication along the *k*th row. □

THEOREM 3. *Given a hyperparallelepiped tile* **L** *and a unimodular reference matrix* **G**, *the size of the cumulative footprint with respect to a set of uniformly intersecting references specified by the reference matrix* **G** *and a set of offset vectors* $\vec{a}_1, \ldots, \vec{a}_R$, *is approximately*

$$|\det \mathbf{D}| + \sum_{k=1}^{d} \left|\det \mathbf{D}_{k \to \hat{a}}\right|$$

where $\hat{a} = \text{spread}_{\mathbf{D}}(\vec{a}_1, \ldots, \vec{a}_R)$ *and* $\mathbf{D}_{k \to \hat{a}}$ *is the matrix obtained by replacing the* k*th row of* **D** *by* $\hat{a}$.

PROOF. As observed earlier, the size of the cumulative footprint is approximately the size of any of the footprints plus the communication across its sides. Clearly the size of any one of the footprints is given by |det **D**|. The rest follows from Lemma 4. □

Finally, as stated earlier, the total communication generated by nonuniformly intersecting sets of references is essentially the sum of the communicating generated by the individual cumulative footprints. Example 8 in Section IV.E discusses an instance of such a computation.

E. Minimizing the Size of the Cumulative Footprint

We now focus on the problem of finding the loop partition that minimizes the size of the cumulative footprint. The overall algorithm is summarized in Table I. The minimization of C, the communication is done using standard optimization algorithms including numerical techniques.

Let us illustrate this procedure through the following two examples.

EXAMPLE 7.

```
Doall (i=1:N, j=1:N, k=1:N)
  A[i,j,k]=B[i-1,j,k+1]+B[i,j+1,k]+B[i+1,j-2,k-3]
EndDoall
```

Here we have two uniformly intersecting sets of references: one for *A* and one for *B*. Let us look at the class corresponding to *B* since it is more instructive. Because *A* has only one reference, whose **G** is unimodular, its footprint size is independent of the loop partition, given a fixed total size of the loop tile, and therefore need not figure in the optimization process. The *G* matrix corresponding to the references to *B* is,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The $\hat{a}$ vector is (2, 3, 4). Consider a rectangular partition **L** = **Λ** given by

$$\begin{bmatrix} L_i & 0 & 0 \\ 0 & L_j & 0 \\ 0 & 0 & L_k \end{bmatrix}$$

In this example, the **D** matrix is the same as the **L** matrix. Because **D** is rectangular, we can apply Proposition 4 in simplifying the derivation of $\hat{a}$. The size of the cumulative footprint for *B* can now be computed according to Theorem 3 as

$$L_i\,L_j\,L_k + 2L_j\,L_k + 3L_i\,L_k + 4L_i\,L_j$$

This expression must be minimized keeping | det **L** | (or the product $L_i\,L_j\,L_k$) a constant. The product represents the area of the loop tile and must be kept constant to ensure a balanced load. The constant is simply the total area of the iteration space divided by *P*, the number of processors. For example, if the loop bounds are *I*, *J*, and *K*, then we must minimize $L_i\,L_j\,L_k + 2L_j\,L_k + 3L_i\,L_k + 4L_i\,L_j$, subject to the constraint $L_i\,L_j\,L_k = IJK/P$.

This optimization problem can be solved using standard methods, for example, using the method of Lagrange multipliers [21]. The size of the cumulative footprint is minimized when L_i, L_j, and L_k are chosen in the proportions 2, 3, and 4, or

$$L_i : L_j : L_k :: 2 : 3 : 4$$

This implies,

$$L_i = (IJK/3P)^{1/3},\ L_j = (3/2)(IJK/3P)^{1/3},\ \textit{and}\ L_k = 2\,(IJK/3P)^{1/3}$$

Abraham and Hudak's algorithm [7] gives an identical partition for this example.

We now use an example to show how to minimize the total number of cache misses when there are multiple uniformly intersecting sets of references. The basic idea here is that the references from each set contribute additively to traffic.

TABLE I
AN ALGORITHM FOR MINIMIZING CUMULATIVE FOOTPRINT SIZE FOR A SINGLE SET OF UNIFORMLY INTERSECTING REFERENCES

Given:	G, offset vectors $\vec{a}_1, \ldots, \vec{a}_R$
Goal:	Find **L** to minimize cumulative footprint size
Procedure:	Write **D** = **LG** Find $b_1, \ldots, b_R = \vec{a}_1 D^{-1}, \ldots, \vec{a}_R D^{-1}$ Find $\ddot{b} = \text{spread}(b_1, \ldots, b_R)$ Then, write $\hat{a} = \ddot{b}\mathbf{D}$ Communication $C = \lvert\det \mathbf{D}\rvert + \sum_{k}\lvert\det \mathbf{D}_{k\to\hat{a}}\rvert$ Finally, find the parameters of **L** that minimize C

For multiple uniformly intersecting sets, add the communication component due to each set and then determine **L** *that minimizes the sum.*

EXAMPLE 8.

```
Doall (i=1:N, j=1:N)
  A(i,j)=B(i-2,j)+B(i,j-1)+C(i+j-1,j)+C(i+j+1,j+3)
EndDoall
```

There are three uniformly intersecting classes of references, one for *B*, one for *C*, and one for *A*. Because *A* has only one reference, its footprint size is independent of the loop partition, given a fixed total size of the loop tile, and therefore need not figure in the optimization process.

For simplicity, let us assume that the tile **L** is rectangular and is given by

$$\begin{bmatrix} L_1 & 0 \\ 0 & L_2 \end{bmatrix}.$$

Because **G** for the references to array *B* is the identity matrix, the **D** = **LG** matrix corresponding to references to *B* is same as **L**, and the $\hat{a}$ vector is *spread*(–2, 0), (0, –1)) = (2, 1). Thus, the size of the corresponding cumulative footprint according to Theorem 3 is

$$\begin{vmatrix} L_1 & 0 \\ 0 & L_2 \end{vmatrix} + \begin{vmatrix} 2 & 1 \\ 0 & L_2 \end{vmatrix} + \begin{vmatrix} L_1 & 0 \\ 2 & 1 \end{vmatrix}.$$

Similarly, **D** for array *C* is

$$\begin{bmatrix} L_1 & 0 \\ L_2 & L_2 \end{bmatrix}.$$

The data footprint **D** is not rectangular even though the loop tile is. Using Definition 13, $\hat{a} = spread_{\mathbf{D}}((-1, 0), (1, 3)) = (4, 3)$, and the size of the cumulative footprint with respect to *C* is

$$\begin{vmatrix} L_1 & 0 \\ L_2 & L_2 \end{vmatrix} + \begin{vmatrix} 4 & 3 \\ L_2 & L_2 \end{vmatrix} + \begin{vmatrix} L_1 & 0 \\ 4 & 3 \end{vmatrix}.$$

The problem of minimizing the size of the footprint reduces to finding the elements of **L** that minimizes the sum of the two expressions above subject to the constraint the area of the loop tile |det *L*| is a constant to ensure a balanced load. For example, if the loop bounds are *I*, *J*, then the constraint is |det **L**| = *I J*/*P*, where *P* is the number of processors.

The total size of the cumulative footprint simplifies to $2L_1L_2 + 4L_1 + 3L_2$. The optimal values for L_1 and L_2 can be shown to satisfy the equation $4L_1 = 3L_2$ using the method of Lagrange multipliers.

V. GENERAL CASE OF G

This section analyzes the size of the footprint and the cumulative footprint for a general **G**, that is, when **G** is not restricted to be unimodular. The computation of the size of the footprint is by case analysis on the **G** matrix.

A. G Is Invertible, but not Unimodular

G is invertible and not unimodular implies that not every integer point in the hyperparallelepiped **D** is an image of an iteration point in **L**. A unit cube in the iteration space is mapped to a hyperparallelepiped of volume equal to |det **G**|. So the size of the data footprint is |det **D**/det **G**| = |det **L**|. When **G** is invertible the size of the data footprint is exactly the size of the loop tile since the mapping is one to one.

Next, the expression for the size of the cumulative footprint is very similar to the one in Theorem 3, except that the data elements accessed are not dense in the data space. That is, the data space is sparse.

LEMMA 5. *Given an iteration space I, a reference matrix* **G**, *and a hyperparallelepiped* $\mathbf{D}_1$ *in the data space, if the vertices of* $\mathbf{D}_1\mathbf{G}^{-1}$ *are in I then the number of elements in the intersection of* $\mathbf{D}_1$ *and the footprint of I with respect to* **G** *is* |det $\mathbf{D}_1$/det **G**|.

PROOF. Clear if one views $\mathbf{D}_1\mathbf{G}^{-1}$ as the loop tile **L**. □

THEOREM 4. *Given a hyperparallelepiped tile* **L**, *and an invertible reference matrix* **G**, *the size of the cumulative footprint with respect to a set of uniformly intersecting references specified by the reference matrix* **G** *and a set of offset vectors* $\vec{a}_1, \ldots, \vec{a}_R$, *is approximately*

$$\frac{|\det \mathbf{D}| + \sum_{k=1}^{d} |\det \mathbf{D}_{k\to\hat{a}}|}{|\det \mathbf{G}|}$$

where $\hat{a} = \text{spread}(\vec{a}_1, \ldots, \vec{a}_R, \mathbf{D})$ *and* $\mathbf{D}_{k\to\hat{a}}$ *is the matrix obtained by replacing the kth row of* **D** *by* $\hat{a}$.

PROOF. Using Lemma 5 one can construct a proof similar to that of Theorem 3. □

Example 2 (repeated below for convenience) possesses a **G** that is invertible, but not unimodular.

```
Doall (i=101:200, j=1:100)
  A[i,j]=B[i+j,i-j-1]+B[i+j+4,i-j+3]
EndDoall
```

For this example, the reference matrix **G** corresponding to array **B** is

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

and the offset vectors are

$$\vec{a}_0 = (0, -1) \text{ and } \vec{a}_1 = (4, 3)$$

Let us find the optimal rectangular partition **L** of the form

$$\begin{bmatrix} L_i & 0 \\ 0 & L_j \end{bmatrix}.$$

The footprint matrix **D** is given by

$$\begin{bmatrix} L_i & L_i \\ L_j & -L_j \end{bmatrix}$$

The offset vectors using **D** as a basis are

$$\vec{b}_0 = \vec{a}_0\mathbf{D}^{-1} = \left(-1/(2L_i), 1/(2L_j)\right),$$
$$\vec{b}_1 = \vec{a}_1\mathbf{D}^{-1} = \left(7/(2L_i), 1/(2L_j)\right).$$

The vector $\hat{b} = (4/L_i, 0)$ and the vector

$$\hat{a} = \hat{b}\mathbf{D} = (4, 4)$$

The size of the cumulative footprint according to Theorem 4 is

$$\frac{\begin{vmatrix} L_i & L_i \\ L_j & -L_j \end{vmatrix} + \begin{vmatrix} L_i & L_i \\ 4 & 4 \end{vmatrix} + \begin{vmatrix} 4 & 4 \\ L_j & -L_j \end{vmatrix}}{\begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix}}$$

which is

$$L_i L_j + 4L_j$$

If we constrain $L_i L_j = 100$ for load balance, we get $L_j = 1$ and $L_i = 100$. This partitioning represents horizontal striping of the iteration space.

B. Columns of G Are Dependent and the Rows Are Independent

We can apply Theorem 4 to compute the size of a footprint when the columns of **G** are dependent, as long as the rows are independent. We derive a **G′** from **G** by choosing a maximal set of independent columns from **G**, such that **G′** is invertible. We can then apply Theorem 4 to compute the size of the footprint as shown in the following example.

EXAMPLE 9. Consider the reference $A[i, 2i, i + j]$ in a doubly nested loop. The columns of the **G** matrix

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

are not independent. We choose **G′** to be

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Now **D′** = **LG′** completely specifies the footprint. The size of the footprint equals |det **D′**| = |det **L**|. If we choose **G′** to be

$$\begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$$

then the size of the footprint is |det **D′**|/2 for the new **D′** since |det **G′**| is now 2. But both expressions evaluate to the same value, |det **L**|, as one would expect.

C. The Rows of G Are Dependent

The rows of **G** are dependent means that the mapping from the iteration space to the data space is many to one. It is hard to derive an expression for the footprint in general when the rows are dependent. However, we can compute the footprint and the cumulative footprint for many special cases that arise in actual programs. In this section we shall look at the common case where the rows are dependent because one or more of the index variables do not appear in the array reference. We shall illustrate our technique with the matrix multiply program shown in Example 10 below. The notation `l$C[i,j]` means that the read-modify-write of `C[i,j]` is atomic.

EXAMPLE 10.

```
Doall (i=0:N, j=0:N, k=0:N)
  l$C[i,j]=l$C[i,j]+A[i,k]+B[k,j]
EndDoall
```

The references to the matrices A, B, and C belong to separate uniformly intersecting references. So the cumulative footprint is the sum of the footprints of each of the references. We will focus on `A[i,k]` and footprint computation for the other references are similar. The **G** matrix for `A[i,k]` is

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

We cannot apply our earlier results to compute the footprint since **G** is a many to one mapping. However, we can find an invertible **G′** such that for every loop tile **L**, there is a tile **L′** such that the number of elements in footprints **LG** and **LG′** are the same. For the current example, **G′** is obtained from **G** by deleting the row of zeros, resulting in a two dimensional identity matrix. Similarly **L′** is obtained from **L** by eliminating the corresponding (second) column of **L**. Now, it is easy to show that the number of elements in footprints **LG** and **LG′** are the same by establishing a one-to-one correspondence between the two footprints. Let us use this method to compute the size of the footprint corresponding to the reference `A[i,k]`. Let us assume that **L** is rectangular to make the computations simpler. Let **L** be

$$\begin{bmatrix} L_i & 0 & 0 \\ 0 & L_j & 0 \\ 0 & 0 & L_k \end{bmatrix}.$$

Now **L′** is

$$\begin{bmatrix} L_i & 0 \\ 0 & 0 \\ 0 & L_k \end{bmatrix}.$$

So the size of the footprint is $L_i L_k$. Similarly, one can show that the size of the other two footprints are $L_i L_j$ and $L_j L_k$. The cumulative footprint is $L_i L_k + L_i L_j + L_j L_k$ which is minimized when L_i, L_j, and L_k are equal.

VI. OTHER SYSTEM ENVIRONMENTS

This section describes how our framework can be used to solve the partitioning problem in a wide range of systems including those with coherent caches, distributed-memory, and non-unit cache line sizes.

A. Coherence-Related Cache Misses

Our analysis presented in the previous section was concerned with minimizing the cumulative footprint size. This process of minimizing the cumulative footprint size not only minimizes the number of first-time cache misses, but the number of coherence-related misses as well. For example, consider the **forall** loop embedded within a sequential loop in Example 11. Here **forall** means that all the reads are done prior to the writes. In other words the data read in iteration t corresponds to the data written in iteration $t - 1$.

EXAMPLE 11.

```
Doseq (t=1:T)
  forall (i=1:N,j=1:N)
    A(i,j)=A(i+1,j)
  EndDoall
EndDoseq
```

For this example, we have

$$\mathbf{G} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Let us attempt to minimize the cumulative footprint for a loop partition of the form

$$\mathbf{L} = \begin{bmatrix} L_i & 0 \\ 0 & L_j \end{bmatrix}$$

The cumulative footprint size is given by

$$L_i L_j + L_j$$

In a load-balanced partitioning, $|\det \mathbf{L}| = L_i L_j$ is a constant, so the $L_i L_j$ term drops out of the optimization. The optimization process then attempts to minimize L_j, which is proportional to the volume of cache coherence traffic, as depicted in Fig. 14.

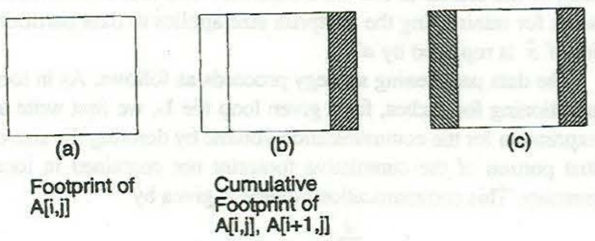


Fig. 14. (a) Footprint of reference $A[i, j]$ for a rectangular L. (b) Cumulative footprint for the references $A[i, j]$ and $A[i + 1, j]$. The hashed region Z represents the increase in footprint size due to the reference $A[i + 1, j]$. (c) The regions X, Y, Z, collectively represent the cumulative footprint for references $A[i, j]$ and $A[i + 1, j]$. Region Z represents the area in the data space shared with the positive neighbor. Region Y represents the area in the data space shared with the negative neighbor.

Let us focus on regions X, Y, and Z in Fig. 14c. As explained in Fig. 13, the processor working on the loop tile to which these regions belong (say, processor P_O) shares a portion of its cumulative footprint with processors working on neighboring regions in the data space. Specifically, region Z is a subtile of the positive neighbor and region Y is a subtile shared with its negative neighbor. Region X, however, is completely private to P_O.

Let us consider the situation after the first iteration of the outer sequential loop. Accesses of data elements within region X will hit in the cache, and thereby incur zero communication cost. Data elements in region Z, however, potentially cause misses because the processor working on the positive neighbor might have previously written into those elements, resulting in those elements being invalidated from P_O's cache. Each of these misses by processor P_O suffers a network round trip because of the need to inform the processor working on its positive neighbor to perform a write-back and then to send the data to processor P_O. Furthermore, if the home memory location for the block is elsewhere, the miss requires an additional network round-trip. Similarly, in region Y, a write by processor P_O potentially incurs two network round trips as well. The two round trips result from the need to invalidate the data block from the cache of the processor working on the negative neighbor, and then to fetch the blocks into P_O's cache.

In any case, the coherence traffic is proportional to the area of the shared region Z, which is equal to the area of the shared region Y, and is given by L_j. So the total communication is minimized by choosing the tile with $L_j = 1$.

B. Effect of Cache Line Size

The effect of cache line sizes can be incorporated easily into our analysis. Because large cache lines fetch multiple data words at the cost of a single miss, one data space dimension will be favored by the cache. Without loss of generality, let us assume that the jth dimension of the data space benefits from larger cache lines. Then, the effect of cache lines of size B can be incorporated into our analysis by replacing each element d_{ij} in the jth column of $\mathbf{D}$ in Theorem 3 by

$$\left\lceil \frac{d_{ij}}{B} \right\rceil$$

to reflect the lower cost of fetching multiple words in the jth dimension of the data space[3], and by modifying the definition of intersecting references to the following.

DEFINITION 14. *Two references $A[\vec{g}_1(\vec{i})]$ and $A[\vec{g}_2(\vec{i})]$ are said to be intersecting if there are two integer vectors $\vec{i}_1$, $\vec{i}_2$ for which $A[\vec{g}_1(\vec{i}_1)] = A[(d_{11}, d_{12}, \ldots)]$ and $A[\vec{g}_2(\vec{i}_2)] = A[(d_{21}, d_{22}, \ldots)]$ such that $A[(\ldots, d_{1(j-1)}, \lceil \frac{d_{1j}}{B} \rceil, \ldots)] = A[(\ldots, d_{2(j-1)}, \lceil \frac{d_{2j}}{B} \rceil, \ldots)]$, where B is the size of a cache line, and the jth dimension in the data space benefits from larger cache lines.*

C. Data Partitioning

In systems in which main memory is distributed with the processing nodes (e.g., see Fig. 5), data partitioning is the problem of partitioning the data arrays into data tiles and the nested loops into loop tiles and assigning the loop tiles to the processing nodes and the corresponding data tiles to memory modules associated with the processing nodes so that a maximum number of the data references made by the loop tiles are satisfied by the local memory module. Our formulation facilitates data partitioning straightforwardly. There are two cases to consider: systems with caches and systems without caches.

C.1. Systems with Caches

The data partitioning strategy in distributed shared-memory systems with caches (Fig. 5a) proceeds as follows. The optimal loop partition $\mathbf{L}$ is first derived by minimizing the cumulative footprint size as described in the previous sections.

Data partitioning requires the additional derivation of the optimal data partition $\mathbf{D}$ for each class of uniformly intersecting references from the optimal loop partition $\mathbf{L}$. We derive the shapes of the data tiles $\mathbf{D}$ for each $\mathbf{G}$ corresponding to a specific class of uniformly intersecting references. A specific data tile is chosen from the footprints corresponding to each

3. We note that the estimate of cumulative footprint size will be slightly inaccurate if the footprint is misaligned with the cache block.

reference in an uniformly intersecting set. In systems with caches, the choice of a specific footprint does not matter, because each data element in the footprint results in a single miss. We then place each loop tile with the data tiles accessed by it on the same processing node.

As an example, let us work out the optimal data partitioning for Example 2. The optimal loop partition for this example was worked out in Section V.A. The optimal **L** was shown to stripe the iteration space horizontally and was given by

$$\begin{bmatrix} 100 & 0 \\ 0 & 1 \end{bmatrix}$$

The corresponding footprint **D** = **LG** represents a diagonal striping of the data space and is given by

$$\begin{bmatrix} 100 & 100 \\ 1 & -1 \end{bmatrix}$$

Thus, for this example, if diagonal tiles of data (as depicted in Fig. 15) are placed in the memory modules close to the processors with the corresponding iteration tiles, cache misses will be satisfied completely within the node. This data partition thus represents a communication-free data partition.

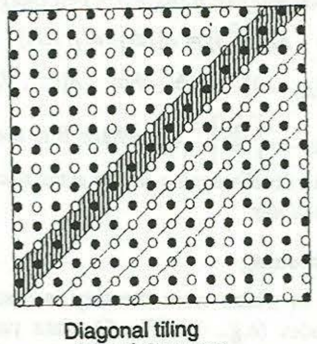


Fig. 15. A communication-free data partition.

Interestingly, because **G** for this example is not unimodular (its determinant is 2), not all data space points are accessed. In the figure, the shaded points represent the untouched data elements.

When a program has multiple loops that access a given data array, the possibility of the loops imposing conflicting data tiling requirements arises. An algorithm for partitioning loops and data in this situation is developed in [15].

C.2. Systems without Caches

The compiler has two options to optimize communication volume in systems without caches. The compiler can choose to make local copies of remote data, or it can fetch remote data each time the data is needed. In the former case, the compiler can use the same partitioning algorithms described in this paper for systems with caches, but it must also solve the data coherence problem for the copied data. This section addresses the latter case.

Although the overall data partitioning strategy remains largely the same as described in the previous section, we must make one change in the footprint size computation to reflect the fact that a given data tile is placed in local memory and data elements from neighboring tiles have to be fetched from remote memory modules each time they are accessed. Because data partitioning for distributed-memory systems without caches (see Fig. 5b) assumes that data from other memory modules is not dynamically copied locally (as in systems with caches), we replace the max - min formulation by the *cumulative spread* a^+ of a set of uniformly intersecting references. That is

$$a^+ = \text{cumulativespread}_{\mathbf{D}}(\vec{a}_1, \ldots, \vec{a}_R) = b^+\mathbf{D},$$

in which the *k*th element of b^+ is given by,

$$b_k^+ = \sum_r \left| \left[b_{r,k} - med_r(b_{r,k}) \right] \right|, \forall k \in 1, \ldots, d,$$

where $\vec{b}_r = \vec{a}_r \mathbf{D}^{-1}, \forall r \in 1, \ldots, R$; and $med_r(b_r, k)$ is the median of the offsets in the *k*th dimension. The rest of our framework for minimizing the footprint size applies to data partitioning if $\hat{a}$ is replaced by a^+.

The data partitioning strategy proceeds as follows. As in loop partitioning for caches, for a given loop tile **L**, we first write an expression for the communication volume by deriving the size of that portion of the cumulative footprint not contained in local memory. This communication volume is given by

$$\sum_{k=1}^{d} \left| \det \mathbf{D}_{k \to a^+} \right|$$

We then derive the optimal **L** to minimize this communication volume. We then derive the optimal data partition **D** for each class of uniformly intersecting references from the optimal loop partition **L** as described in the previous section on systems with caches. A specific data tile is chosen from the footprints corresponding to each reference in an uniformly intersecting set. In systems without caches, because a single data element might have to be fetched multiple times, the choice of a specific data footprint does matter. A simple heuristic to maximize the number of local accesses is to choose a data tile whose offsets are the medians of all the offsets in each dimension. We can show that using a median tile is optimal for one-dimensional data spaces, and close to optimal for higher dimensions. However, a detailed description is beyond the scope of this paper. We then place each loop tile with the corresponding data tiles accessed by it on the same processor.

VII. Implementation and Results

This paper presents cumulative footprint size measurements from an algorithm simulator and execution time measurements from an actual compiler implementation on a multiprocessor.

A. Algorithm Simulator Experiments

We have written a simulator of partitioning algorithms that measures the exact cumulative footprint size for any given hyperparallelepiped partition. The simulator also presents analytically computed footprint sizes using the formulation presented in Theorem 3.

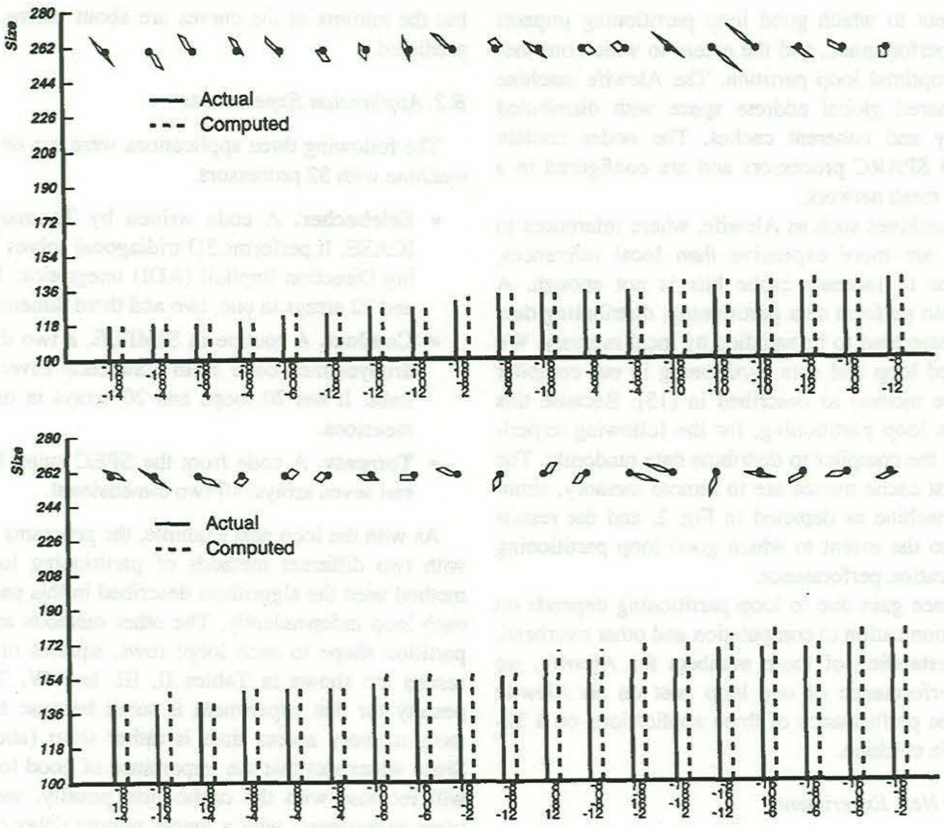


Fig. 16. Actual and computed footprint sizes for several loop partitions.

We present in Fig. 16 algorithm simulator data showing the communication volume for array **B** in Example 3 (repeated below for convenience) resulting from a large number of loop partitions (with tile size 96) representing both parallelograms and rectangles. The abscissa is labeled by the **L** matrix parameters of the various loop partitions, and the parallelogram shape is also depicted above each histogram bar.

```
Doall (i=1:N, j=1:N)
  A[i,j]=B[i,j]+B[i+1,j-2]+B[i-1,j+1]
EndDoall
```

The example demonstrates that the analytical method yields accurate estimates of cumulative footprint sizes. The estimates are higher than the measured values when the partitions are mismatched with the offset vectors due to the overlapping subtile approximation described in Section IV.D. We can also see that the difference between the optimal parallelogram partition and a poor partition is significant. The differences become even greater if bigger offsets are used. This example also shows that rectangular partitions do not always yield the best partition.

B. Experiments on the Alewife Multiprocessor

We have also implemented some of the ideas from our framework in a compiler for the Alewife machine [22] to understand the extent to which good loop partitioning impacts end application performance, and the extent to which our theory predicts the optimal loop partition. The Alewife machine implements a shared global address space with distributed physical memory and coherent caches. The nodes contain slightly modified SPARC processors and are configured in a two-dimensional mesh network.

For NUMA machines such as Alewife, where references to remote memory are more expensive than local references, partitioning loops to increase cache hits is not enough. A compiler must also perform data partitioning, distributing data so that cache misses tend to be satisfied by local memory. We have implemented loop and data partitioning in our compiler using an iterative method as described in [15]. Because this paper focuses on loop partitioning, for the following experiments we caused the compiler to distribute data randomly. The effect is that most cache misses are to remote memory, simulating a UMA machine as depicted in Fig. 2, and the results offer insights into the extent to which good loop partitioning affects end application performance.

The performance gain due to loop partitioning depends on the ratio of communication to computation and other overhead. To get an understanding of these numbers for Alewife, we measured the performance of one loop nest on an Alewife simulator, and the performance of three applications on a 32-processor Alewife machine.

B.1. Single Loop Nest Experiment

The following loop nest was run on a simulator of a 64 processor Alewife machine:

```
Doall (i=0:255, j=4:251)
  A[i,j]=A[i-1,j]+B[i,j+4]+B[i,j-4]
EndDoall
```

The **G** matrix for the above loop nest is the 2×2 identity matrix, and the offset vectors are $\vec{a}_1 = (0, 0)$, $\vec{a}_2 = (-1, 0)$, $\vec{b}_1 = (0, 4)$, and $\vec{b}_2 = (0, -4)$. Each array was 512 elements (words) on a side. The cache line size is four words, and the arrays are stored in row-major order.

Using the algorithms in this paper, and taking the four-word cache line size into account, the compiler chose a rectangular loop partition and determined that the optimal partition has an aspect ratio of 2 : 1. The compiler then chose the closest aspect ratio (1 : 1) that also achieves load balance for the given problem size and machine size, which results in a tile size of 64×64 iterations. We also ran the loop nest using suboptimal partitions with tile dimensions ranging from 8×512 to 512×8. This set of executions is labeled run A in Fig. 17. We ran a second version of the program using a different set of offset vectors that give an optimal aspect ratio of 8 : 1 (run B). This results in a desired tile size between 256×16 and 128×32 with the compiler choosing 256×16.

Fig. 17 shows the running times for the different tile sizes, and demonstrates that the compiler was able to pick the optimal partitions for both cases. There is some noise in these figures because there can be variation in the cost of accessing the memory that is actually shared due to cache coherence actions, but the minima of the curves are about where the framework predicted.

B.2. Application Experiments

The following three applications were run on a real Alewife machine with 32 processors.

- **Erlebacher.** A code written by Thomas Eidson, from ICASE. It performs 3D tridiagonal solves using Alternating Direction Implicit (ADI) integration. It has 40 loops and 22 arrays in one, two and three dimensions.
- **Conduct.** A routine in SIMPLE, a two dimensional hydrodynamics code from Lawrence Livermore National Labs. It has 20 loops and 20 arrays in one and two dimensions.
- **Tomcatv.** A code from the SPEC suite. It has 12 loops and seven arrays, all two dimensional.

As with the loop nest example, the programs were compiled with two different methods of partitioning loops. The *auto* method used the algorithms described in this paper to partition each loop independently. The other methods assigned a fixed partition shape to each loop: rows, squares or columns. The results are shown in Tables II, III, and IV. The cache-miss penalty for this experiment is small because the Alewife remote memory access time is rather short (about 40 cycles). Since we expect that the importance of good loop partitioning will increase with the cache-miss penalty, we also ran two other experiments with a longer remote delay of 100 and 200 cycles. Alewife allows longer delays to be synthesized by a combination of software and hardware mechanisms.

These results show that the choice of partitioning parameters affects performance significantly. In all cases, the partitioner was able to discover the best partition. In two of the applications, the compiler's partition choice resulted in a small improvement over squares. In Tomcatv, the compiler chose the same square partition for each loop, resulting in no improvement over the fixed square partition. The performance gains over squares for all of these programs are modest because the offsets in most of the references in the three applications are similar.

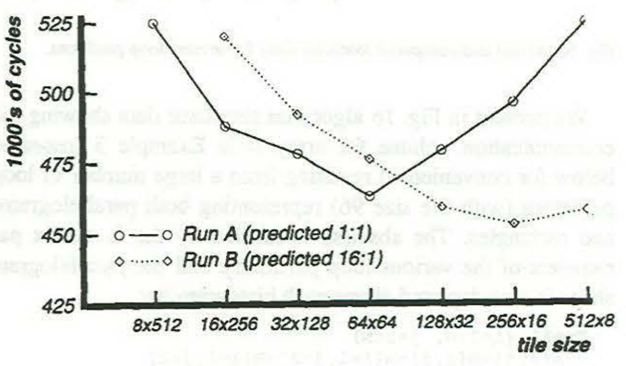


Fig. 17. Running times in 1,000s of cycles for different aspect ratios on 64 processors.

TABLE II
EXECUTION TIME IN MCYCLES FOR ERLEBACHER (N = 64)

Delay	*auto*	rows	squares	columns
40 cycles	27.0	27.3	28.6	28.2
100 cycles	30.4	31.4	31.2	31.3
200 cycles	34.0	35.2	36.4	36.8

TABLE III
EXECUTION TIME IN MCYCLES FOR CONDUCT (N = 768)

Delay	*auto*	rows	squares	columns
40 cycles	67.2	71.2	71.4	71.2
100 cycles	85.4	91.2	91.8	90.8
200 cycles	111.4	118.2	117.1	117.5

TABLE IV
EXECUTION TIME IN MCYCLES FOR TOMCATV (N = 1,200)

Delay	*auto*	rows	squares	columns
40 cycles	104	127	100	113
100 cycles	125	152	122	138
200 cycles	154	188	154	174

VIII. CONCLUSIONS

The performance of cache-coherent systems is heavily predicated on the degree of temporal locality in the access patterns of the processor. If each block of data is accessed a number of times by a given processor, then caches will be effective in reducing network traffic. Loop partitioning for cache-coherent multiprocessors strives to achieve precisely this goal.

This paper presented a theoretical framework to derive the parameters of iteration-space partitions of the do loops to minimize the communication traffic in multiprocessors with caches. The framework allows the partitioning of doall loops into optimal hyperparallelepiped tiles where the index expressions in array accesses can be any affine function of the indices. The same framework also yields optimal loop and data partitions for multicomputers with local memory.

Our analysis uses the notion of uniformly intersecting references to categorize the references within a loop into classes that will yield cache locality. A theory of data footprints is introduced to capture the combined set of data accesses made by the references within each uniformly intersecting class. Then, an algorithm to compute precisely the total size of the data footprint for a given loop partition is presented. Once an expression for the total size of the data footprint is obtained, standard optimization techniques can be applied to minimize the size of the data footprint and derive the optimal loop partitions.

Our framework discovers optimal partitions in many more general cases than those handled by previous algorithms. In addition, it correctly reproduces results from loop partitioning algorithms for certain special cases previously proposed by other researchers.

The framework, including both loop and data partitioning for cache-coherent distributed shared-memory, has been implemented in the compiler system for the Alewife multiprocessor.

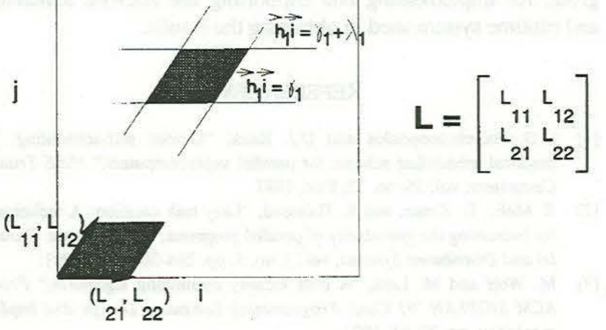
APPENDICES

A. A Formulation of Loop Tiles Using Bounding Hyperplanes

A specific hyperparellelepiped loop tile is defined by a set of bounding hyperplanes. Similar formulations have also been used earlier [6].

DEFINITION 15. *Given an l dimensional loop nest $\vec{i}$, each tile of a hyperparallelepiped* loop partition *is defined by the hyperplanes given by the rows of the $l \times l$ matrix* **H** *and the column vectors $\vec{\gamma}$ and $\vec{\lambda}$ as follows. The parallel hyperplanes are $\vec{h}_j\vec{i} = \gamma_j$ and $\vec{h}_j\vec{i} = \gamma_j + \lambda_j$, for $1 \leq j \leq l$. An iteration belongs to this tile if it on or inside the hyperparallelepiped.*

When loop tiles are assumed to be homogeneous except at the boundaries of the iteration space, the partitioning is completely defined by specifying the tile at the origin, namely ($\mathbf{H}$, $\vec{0}$, $\vec{\lambda}$), as inicated in Fig. 18. For notational convenience, we denote the tile at the origin as **L**.



$\vec{h}_1\vec{i} = \gamma_1 + \lambda_1$
$\vec{h}_1\vec{i} = \gamma_1$
j
$\mathbf{L} = \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix}$
(L_{11}, L_{12})
(L_{21}, L_{22})
i

Fig. 18. Iteration space partitioning is completely specified by the tile at the origin.

DEFINITION 16. *Given the tile* ($\mathbf{H}$, $\vec{0}$, $\vec{\lambda}$) *at the origin of the hyperparallelepiped partition, let* $\mathbf{L} = \mathbf{L}(\mathbf{H}) = \Lambda(\mathbf{H}^{-1})^T$, *where Λ is a diagonal matrix with $\Lambda_{ii} = \lambda_i$. We refer to the tile by the* **L** *matrix, as* **L** *completely defines the tile at the origin. The rows of* **L** *specify the vertices of the tile at the origin.*

B. Synchronization References

Sequential do loops can often be converted to parallel do loops by introducing fine-grain data-level synchronization to enforce data dependencies or mutual exclusion. The cost of synchronization can be approximately modeled as slightly more expensive communication [14]. For example, in the Alewife system the inner loop of matrix multiply can be written using fine-grain synchronization in the form of the loop in Example 12.

EXAMPLE 12.
```
Doall (i=1:N, j=1:N, k=1:N)
   1$C[i,j]=1$C[i,j]+A[i,k]+B[k,j]
EndDoall
```

In the code segment in Example 12, the "1$" preceding the C matrix references denote atomic accumulates. Accumulates into the C array can happen in any order, just that each accumulate action must be atomic. Such synchronization reads or writes are both treated as writes by the coherence system. Similar linguistic constructs are also present in Id [23] and in a variant of Fortran used on the HEP [24].

ACKNOWLEDGMENTS

This research is supported by Motorola Cambridge Research Center and by National Science Foundation grant #MIP-9012773. Partial support has also been provided by DARPA contract #N00014-87-K-0825, in part by a National Science Foundation Presidential Young Investigator Award. We are grateful to Rajeev Barua for pointing out an error in an earlier formulation of the footprint size and for extending our compiler implementation to include general affine index expressions and data partitioning. Gino Maa helped define and implement the compiler system and its intermediate form. Andrea Carnevali suggested the simple proof for Theorem 1 on lattices sketched in this paper. We acknowledge the contributions of the Alewife group for implementing and supporting the Alewife simulator and runtime system used in obtaining the results.

REFERENCES

[1] C.D. Polychronopoulos and D.J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Trans. Computers*, vol. 36, no. 12, Dec. 1987.

[2] E. Mohr, D. Kranz, and R. Halstead, "Lazy task creation: A technique for increasing the granularity of parallel programs," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 3, pp. 264-280, July 1991.

[3] M. Wolf and M. Lam, "A data locality optimizing algorithm," *Proc. ACM SIGPLAN '91 Conf. Programming Language Design and Implementation*, pp. 30-44, 1991.

[4] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation," *J. Parallel and Distributed Computing*, vol. 5, pp. 587-616, 1988.

[5] H.S. Stone and D. Thiebaut, "Footprints in the cache," *Proc. ACM SIGMETRICS 1986*, pp. 4-8, May 1986.

[6] F. Irigoin and R. Triolet, "Supernode partitioning," *15th Symp. Principles of Programming Languages (POPL XV)*, pp. 319-329, Jan. 1988.

[7] S.G. Abraham and D.E. Hudak, "Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 3, pp. 318-328, July 1991.

[8] J. Ramanujam and P. Sadayappan, "Complie-time techniques for data distribution in distributed memory machines," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 4, pp. 472-482, Oct. 1991.

[9] J.M. Anderson and M.S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines," *Proc. SIGPLAN '93 Conf. Programming Languages Design and Implementation*, ACM, June 1993.

[10] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 2, pp. 179-193, Mar. 1992.

[11] R. Schreiber and J. Dongarra, "Automatic blocking of nested loops," Technical report, RIACS, NASA Ames Research Center and Oak Ridge Nat'l Laboratory, May 1990.

[12] J. Ferrante, V. Sarkar, and W. Thrash, "On estimating and enhancing cache effectiveness," *Lecture Notes in Computer Science: Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds., pp. 328-341, Springer-Verlag, Aug. 1991.

[13] J. Ramanujam and P. Sadayappan, "Tiling multidimensional iteration spaces for nonshared memory machines," *Proc. Supercomputing '91*, IEEE CS Press, 1991.

[14] G.N. Srinivasa Prasanna, A. Agarwal, and B.R. Musicus, "Hierarchical compilation of macro dataflow graphs for multiprocessors with local memory," *IEEE Trans. Parallel and Distributed Systems*, July 1994.

[15] R. Barua, D. Kranz, and A. Agarwal, "Global partitioning of parallel loops and data arrays for caches and distributed memory in multiprocessors," Technical Memo MIT-LCS TM-538, Massachusetts Institute of Technology, 1995.

[16] A. Agarwal, J.V. Guttag, C.N. Hadjicostis, and M.C. Papaefthymiou, "Memory assignment for multiprocessor caches through grey coloring," *PARLE '94 Parallel Architectures and Languages Europe*, pp. 351-362, Springer Verlag Lecture Notes in Computer Science 817, July 1994.

[17] M. Lam, E.E. Rothbert, and M.E. Wolf, "The cache performance and optimizations of blocked algorithms," *Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pp. 63-74, ACM, Apr. 1991.

[18] A. Carnevali, V. Natarajan, and A. Agarwal, "A relationship between the number of lattice points within hyperparallelepipes and their volume," Motorola Cambridge Research Center, in preparation, Aug. 1993.

[19] G. Strang, *Linear Algebra and Its Applications*, third edition. San Diego, Calif.: Harcourt Brace Jovanovich, 1988.

[20] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons, 1990.

[21] G. Arfken, *Mathematical Methods for Physics*. Academic Press, 1985.

[22] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife machine: Architecture and performance," *Proc. 22nd Ann. Int'l Symp. Computer Architecture (ISCA '95)*, June 1995.

[23] P.S. Barth, R.S. Nikhil, and Arvind, "M-structures: Extending a parallel, nonstrict, functional language with state," *Proc. Fifth ACM Conf. Functional programming Languages and Computer Architecture*, Aug. 1991.

[24] B.J. Smith, "Architecture and applications of the HEP multiprocessor computer system," *Society Photo-Optical Instrumentation Engineers*, vol. 298, pp. 241-248, 1981.

Anant Agarwal received a BTech degree from the Indian Institute of Technology, Madras, India, in 1982, and a PhD from Stanford University in 1987. He is an associate professor of electrical engineering and computer science with the Laboratory for Computer Science at MIT, where he led the Alewife multiprocessors system project. Dr. Agarwal is also chief scientist with Virtual Machine Works, Inc., a company he cofounded that focuses on compilation technologies for multiFPGA systems. While at Stanford, he participated in the MIPS and MIPS-X projects. Dr. Agarwal's current research interests include the design of scalable multiprocessor systems and reconfigurable computing using FPGAs.

David A. Kranz received the BA degree from Swarthmore College in 1981, and the PhD degree from Yale University, where he worked on high-performance compilers for Scheme and applicative languages, in 1988. He has been a research associate with the MIT Laboratory for Computer Science since 1987 and was software architect of the Alewife project. Dr. Kranz's research interests are in programming language design and implementation for parallel computing.

Venkat Natarajan received a PhD in computer science from Rutgers University in 1981. He was an assistant professor in the Computer Science Department at Tufts University prior to joining Compass, Inc., in Massachusetts, where he was a codesigner of the Fortran 90 compiler for the MasPar MP-1. Dr. Natarajan has been a research scientist at the Motorola Cambridge Research Center since June 1990 and is also a research affiliate of the Laboratory for Computer Science at MIT. His research interests include compilers, neural networks, performance evaluation, and parallel computation.